TMS320C6201/6701 Evaluation Module Technical Reference

Literature Number: SPRU305
December 1998

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 document provides technical information that describes the TMS320C6x ('C6x) evaluation module (EVM) functionality.

It is assumed that you are already in possession or have access to the companion *TMS320C6201/C701 EVM User's Guide* SPRU269C, for a thorough review of how to install and operate the 'C6x EVM. For up-to-date information on the 'C6x EVM, as well as related products, visit the 'C6000 website at:
http://www.ti.com/sc/docs/dsps/tools/c6000/index.htm.

How to Use This Manual

This reference provides the following types of information about the 'C6x EVM:

- Chapter 1 describes the theory of operation for the 'C6x EVM hardware, including key component identification, detailed information about each functional area, and interface descriptions.
- Chapter 2 describes the 'C6x EVM host software utilities.
- Chapter 3 describes the DSP host support software. It includes a complete API reference that allows you to write your own applications for the 'C6x EVM and also includes example applications that use the API calls.

❑ **Reference material**, consisting of Appendixes A through E, provides quick reference information for the 'C6x EVM:

- Appendix A provides the EVM connector pinouts.
- Appendix B contains the EVM schematics.
- Appendix C provides the EVM complex programmable logic device (CPLD) equations.
- Appendix D summarizes the EVM 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, PDB[7:0] represents the 8 peripheral data bus pins (PDB7 through PDB0) on a device.

- The TMS320C6x family of devices is referred to as the ’C6x. The following abbreviations are used in this manual for TI devices on the ’C6x EVM:

Abbreviation	Device Definition
’C6201	TMS320C6201
’ALB16244	SN74ALB16244
’CBT3257	SN74CBT3257
’CBTD3384	SN74CBTD3384
’LVT125	SN74LVT125
’LVTH162245	SN74LVTH162245

Information About Cautions and Warnings

This book contains cautions and warnings.

This is an example of a caution statement.

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

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 ’C6x 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.

TMS320C6x 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 ’C6x 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.

TMS320C6x Optimizing C Compiler User’s Guide (literature number SPRU187) describes the ’C6x C compiler and the assembly optimizer. This C compiler accepts ANSI standard C source code and produces assembly language source code for the ’C6x generation of devices. The assembly optimizer helps you optimize your assembly code.

TMS320C62x/C67x CPU and Instruction Set Reference Guide (literature number SPRU189) describes the ’C62x/C67x 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.

TMS320C62x/C67x Programmer’s Guide (literature number SPRU198) describes ways to optimize C and assembly code for the TMS320C62x/C67x DSPs and includes application program examples.

TMS320C62x/C67x Technical Brief (literature number SPRU197) gives an introduction to the ’C62x/C67x 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.

TMS320C6201 Digital Signal Processor Data Sheet (literature number SPRS051) describes the features of the TMS320C6201 fixed-point DSP and provides pinouts, electrical specifications, and timings for the device.

TMS320C6701 Digital Signal Processor Data Sheet (literature number SPRS067) describes the features of the TMS320C6701 floating-point DSP and provides pinouts, electrical specifications, and timings for the device.

TMS320C6201/6701 Evaluation Module Users Guide, (literature number SPRU269C) can be accessed from the following URL:

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

Related Documentation

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.

Trademarks

Obtaining Technical Support

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

- ❑ Assembly number of your ’C6x EVM (D600830-0001) located on the bottom side of the board
- ❑ Revision number of your ’C6x EVM 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 ’C6x EVM confidence test utility results that identifies potential problems and other revision numbers (software, EEPROM, CPLD). Chapter 3, *Running the Board Confidence Test*, in *TMS320C6201/6701 EVM Users Guide* 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 your 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.

  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 TMS320C6x EVM is shared with another device. If it is, this is probably the problem. See Appendix A, *Troubleshooting*, in *TMS320C6201/6701 EVM Users Guide* for the corrective action.

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	(281) 274-2320	Fax: (281) 274-2324	Email: dsph@ti.com
DSP Modem BBS	(281) 274-2323		

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

TMS320C6x EVM Hardware

This chapter describes the TMS320C6x EVM 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.

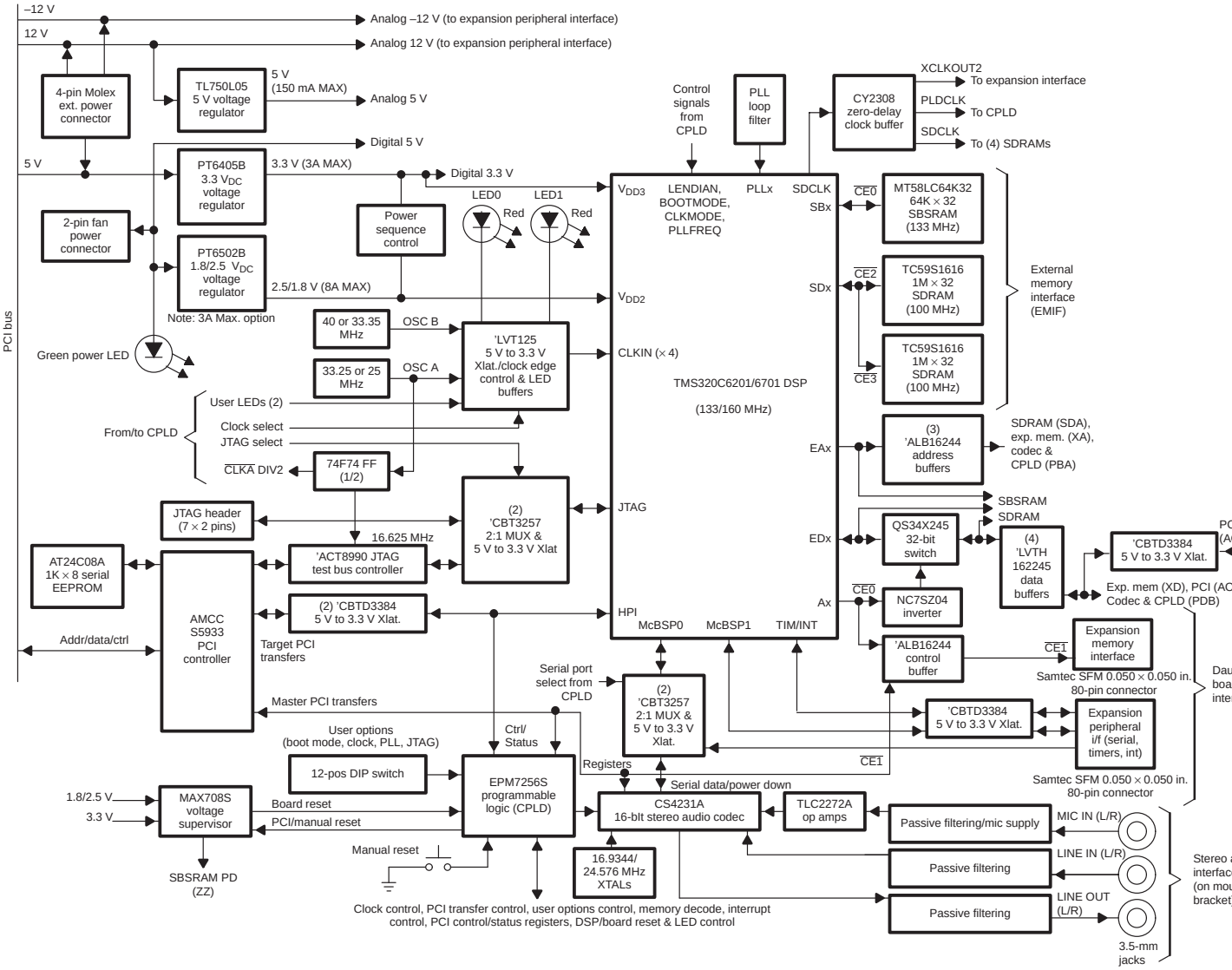
The hardware can be divided into 12 functional areas. Each of these areas is discussed in detail in this chapter.

Topic **Page**

1.1 TMS320C6x EVM Hardware Detailed Block Diagram

Figure 1–1 provides a detailed block diagram of the ’C6x EVM hardware. It identifies the key components described in this chapter and shows how they interface to each other.

Figure 1–1. TMS320C6x EVM Detailed Block Diagram



1.2 TMS320C6201/6701 DSP

The TMS320C6201/6701 DSP is the brain of the 'C6x EVM. The 'C6201/6701 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
  - 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

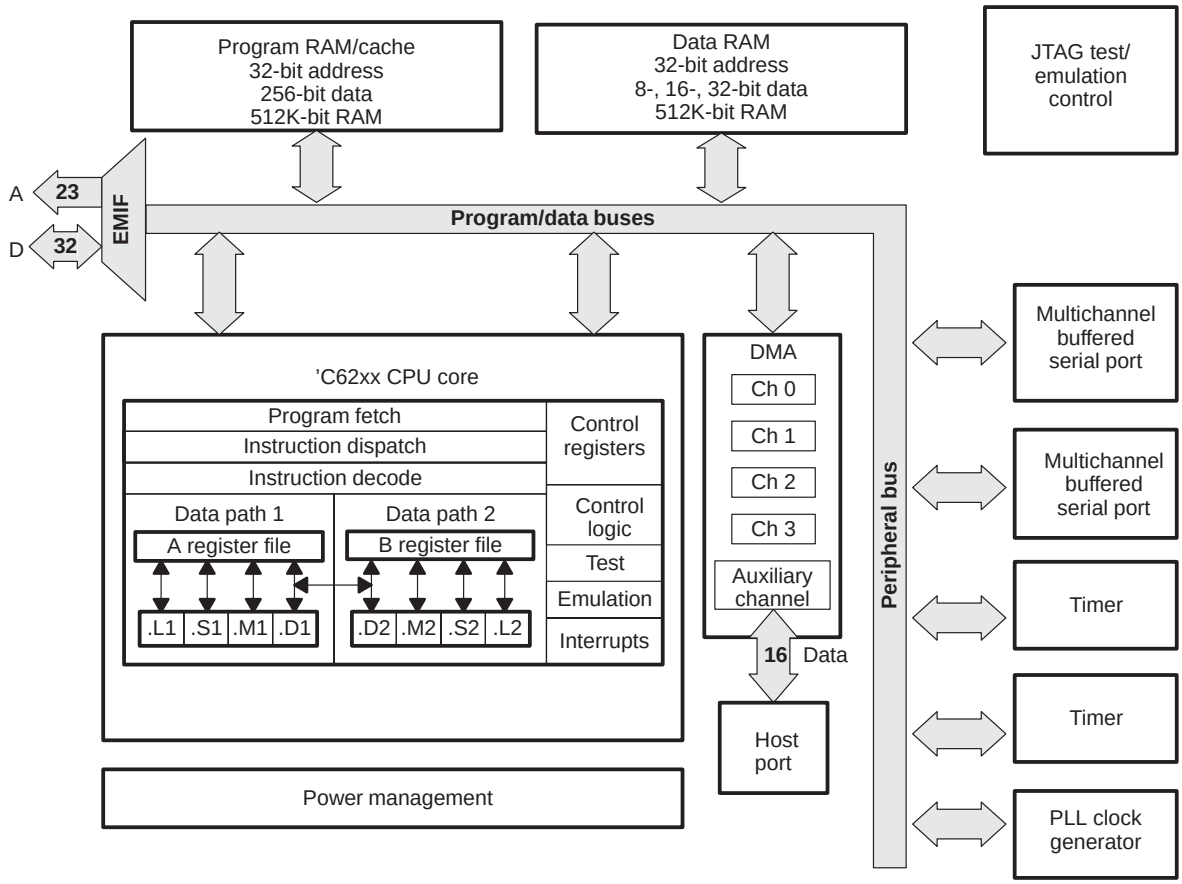
- Two general-purpose timers
- Multiply-by-4 phase locked loop (PLL) and multiply-by-1 PLL-bypass options
- 1M-bit on-chip memory (2K × 256 bits of program memory/64K bytes of data memory)

❑ Miscellaneous

- IEEE-1149.1 (JTAG) boundary-scan compatible for emulation and test support
- 352-lead ball grid array (BGA) package
- 0.25-micron/5-level metal process with CMOS technology
- 3.3-V I/O and 2.5-V or 1.8-V internal core voltages

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

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



The C6x EVM may contain either a ’6201 (fixed-point) or ’6701 (floating-point) DSP and may be operated at different frequencies. The following text describes the frequencies in terms of their relationship to a specific oscillator (speed × 1/2, speed × 4, etc.). Table 1–1 shows the values for the oscillators on the EVM.

Table 1–1. Quad Clock Support Frequencies

	OSC A		OSC B	
	x1	x4	x1	x4
’C6201 EVM	33.25 MHz	133 MHz	40 MHzB	160 MHz
’C6701 EVM	25 MHz	100 MHz	33.25 MHz	133 MHz

Because the ’C6x EVM is a hardware reference design intended to provide you with maximum flexibility, it supports all of the DSP’s external interfaces. The ’C6x EVM uses the PLL clock generator interface to support four clock rates: OSC A × 1, OSC B × 1, OSC A × 4, and OSC B × 4. 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, stereo audio codec, 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 stereo audio codec and/or a daughterboard connected to the expansion peripheral interface.

Some versions of the C6x EVM use an extremely low-profile, integrated heat sink/fan cools the ’C6201/6701 DSP for maximum performance and reliability. This 7.5-mm-high unit is designed for the ’C6201’s 35-mm × 35-mm BGA package. It requires 5 V_{DC} at 100 mA and reduces the temperature rise on the device by 50–80% to handle the maximum dissipation of the DSP in a PCI environment that has little or no air flow. The ’C6x EVM provides a small 2-pin power connector (J9) to supply the fan’s power.

1.3 DSP Clocks

The ’C6x EVM provides quad DSP clock support, which allows you to run benchmarks using different clocks to determine optimal performance for a particular application.

The EVM uses two half-size oscillators, along with a TI SN74LVT125 quad bus buffer device, to provide OSC A and OSC B CLKIN signals to the ’C6201/6701. 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 EVM provides user options for selecting the CLKIN frequency (OSC A or OSC B) and the clock mode (multiply-by-1 or multiply-by-4 as shown in Table 1–2). 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 1–2. DSP Clock Summary

Clock Source	Clock Mode	SSCLK (1/2 rate/1× rate)	SDCLK
OSC A	Multiply-by-1	16.625/OSC A MHz	1/2 OSC A
OSC B	Multiply-by-1	20/OSC B	1/2 OSC B
OSC A	Multiply-by-4	OSC A × 2/OSC A × 4	OSC A × 2
OSC B	Multiply-by-4	80 MHz†	OSC B × 2

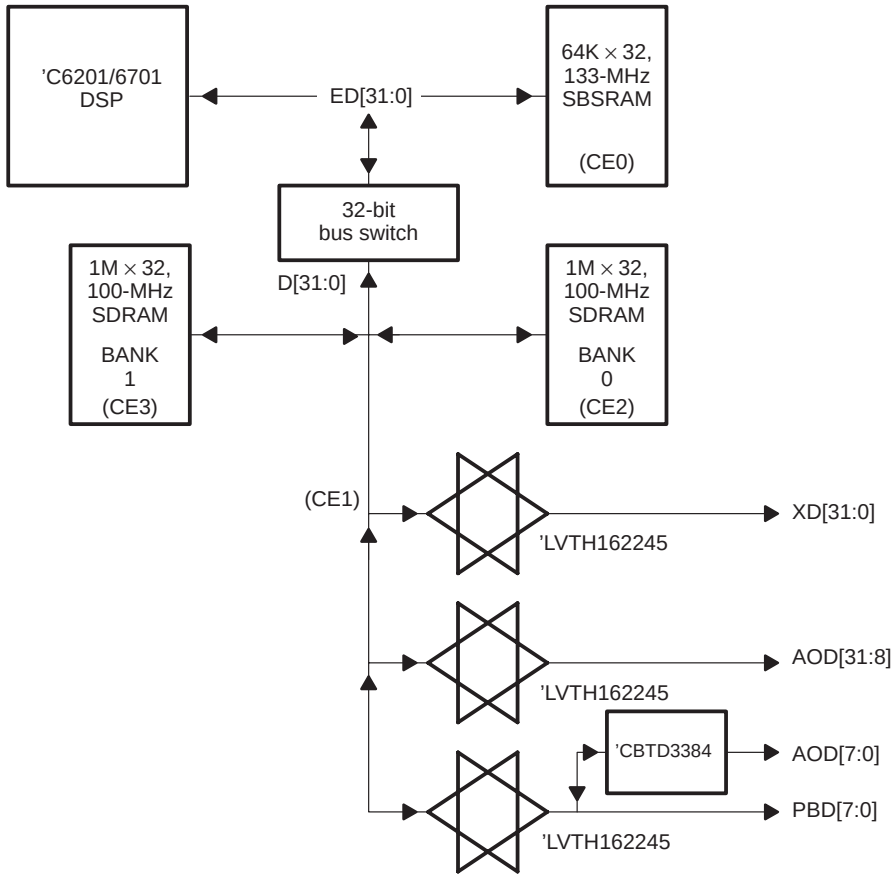
† 1× SSCLK rate is invalid when CLKOUT1 is 160 MHz.

1.4 External Memory

The ’C6x EVM provides one bank of 64K × 32-bit words of 7.5-ns (133-MHz) SBSRAM and two banks of 1M × 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 ’LVTH162245 5-V-tolerant buffers are used.

The EMIF address and data busses are buffered (except to the SBSRAM) to preserve signal integrity, limit loading on the ’C6201/6701 outputs, and provide the necessary drive to meeting timing requirements. A 32-bit bus switch device is used to isolate the rest of the data bus during SBSRAM accesses. This isolation preserves signal integrity and allows the EVM to run the SBSRAM at the full 133-MHz bus speed. Very fast line drivers are used to buffer the address and control signals, which limits the loading on the DSP’s outputs. Fast data transceivers are used to provide voltage translation and the necessary drive to a daughterboard. Figure 1–3 and Figure 1–4 show the configuration of the EMIF data and address bus buffering, respectively.

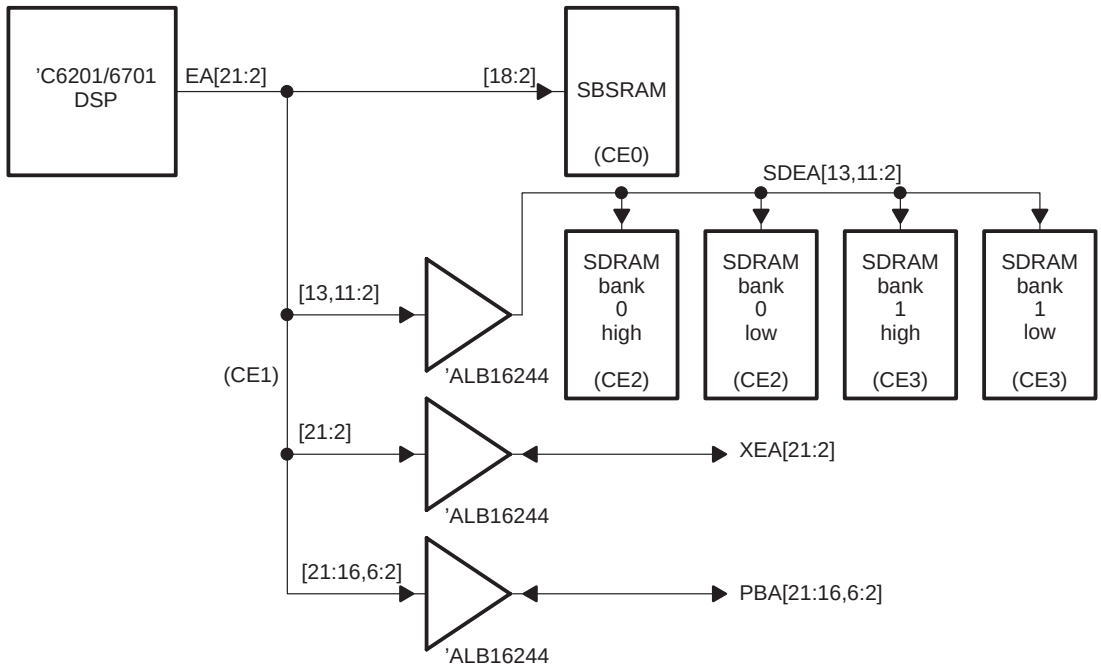
Figure 1–3. EMIF Data Bus Topology



Legend: ED DSP EMIF data
XD Expansion memory data
AOD PCI add-on data
PBD Peripheral bus data

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



Legend: EA DSP EMIF address
SDEA SDRAM access
XEA Expansion memory address
PBA Peripheral bus address

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

1.4.1 SBSRAM

The EVM provides 64K × 32-bit words of SBSRAM. The SBSRAM used on the EVM 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 64K × 32-bit words of memory in a single 100-pin thin quad flat pack (TQFP) package. The pinout of this package is the same for the 128K × 32-bit word device, but it uses one extra address line. The EVM layout supports the use of 128K × 32-bit devices.

The SBSRAM device can be clocked by the 'C6201/6701 at the CPU clock speed when operating at 133 MHz, one-half the clock speed when operating at 160 MHz, or at either the same as or one-half the clock speed when operating in the multiply-by-1 clock mode (OSC A and OSC B). 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 EVM.

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. With this power management feature, the maximum current is reduced from 110 mA to 0.5 mA. Data in the SBSRAM is retained during the snooze mode.

1.4.2 SDRAM

The EVM provides two banks of 1M $\times$ 32-bit words of SDRAM. Each bank is comprised of two 512K $\times$ 2 banks $\times$ 16-bit devices. The 3.3-V SDRAM used on the ’C6x EVM 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 4M bytes of the 16M-byte address space. One of the banks of SDRAM (CE3) resides on the back side of the board to optimize signal routing and signal integrity.

The SDRAM devices are always clocked at one-half the CPU clock rate. This means that when the DSP core runs at OSC A $\times$ 4, the SDRAM runs at 66.5 MHz (15 ns), and when the DSP core runs at OSC B $\times$ 4, the SDRAM runs at 80 MHz (12.5 ns). The EVM’s SDRAM devices are rated for 100-MHz operation.

The four SDRAM devices have dedicated point-to-point clocks that are driven by a clock driver device. The clock driver provides near zero-delay PLL buffering of the DSP’s SDCLK. Two clock driver outputs are routed to the expansion peripheral interface connector and the CPLD. Address and control signals are buffered with TI ’ALB16244 buffers to manage the loading of these ’C6201 signals properly.

The SDRAM must be refreshed periodically to maintain its data. The ’C6x EVM 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 EVM'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. With this power management feature, the maximum per-device current is reduced from 120 mA to 3 mA.

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 EVM provides an asynchronous expansion memory interface connector (J6) 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 EVM's stereo audio codec, 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 (J7). 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 'LVTH 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 EVM to be autobooted at power up and reset with an application stored in nonvolatile memory. The EVM can operate in this mode inside the PC or, more typically, in an external operating environment.

1.4.4 DSP Memory Maps

Table 1–3 and Table 1–4 show the 'C6x EVM DSP memory maps for MAP 0 and MAP 1 modes, respectively.

Table 1–3. TMS320C6x EVM DSP Memory Map (MAP 0)

Start Address	End Address	External Memory Space	Size (Bytes)	Description
00000000	0003FFFF	CE0	256K	SBSRAM
00040000	00FFFFFF	CE0	16M – 256K	Unused
01000000	012FFFFF	CE1	3M	Asynchronous expansion memory
01300000	0130003F	CE1	64	PCI add-on registers
01300040	0130FFFF	CE1	64K – 64	Unavailable
01310000	01310003	CE1	4	PCI FIFO
01310004	0131FFFF	CE1	64K – 4	Unavailable
01320000	0132000F	CE1	16	Audio codec registers
01320010	0132FFFF	CE1	64K – 16	Unavailable
01330000	0137FFFF	CE1	320K	Reserved
01380000	0138001F	CE1	32	DSP control/status registers
01380020	0138FFFF	CE1	64K – 32	Unavailable
01390000	0x13FFFF	CE1	448K	Reserved
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	023FFFFF	CE2	4M	SDRAM (bank 0) or optional asynchronous expansion memory
02400000	02FFFFFF	CE2	12M	Reserved
03000000	033FFFFF	CE3	4M	SDRAM (bank 1) or optional asynchronous expansion memory
03400000	03FFFFFF	CE3	12M	Reserved
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 1–4. TMS320C6x EVM 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	0043FFFF	CE0	256K	SBSRAM
00440000	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	Unavailable
01710000	01710003	CE1	4	PCI FIFO
01710004	0171FFFF	CE1	64K – 4	Unavailable
01720000	0172000F	CE1	16	Audio codec registers
01720010	0172FFFF	CE1	64K – 16	Unavailable
01730000	0177FFFF	CE1	320K	Reserved
01780000	0178001F	CE1	32	DSP control/status registers
01780020	0178FFFF	CE1	64K – 32	Unavailable
01790000	017FFFFF	CE1	448K	Reserved
01800000	01BFFFFF	N/A	4M	Internal peripherals
01C00000	01FFFFFF	N/A	4M	Reserved
02000000	023FFFFF	CE2	4M	SDRAM (bank 0) or optional asynchronous expansion memory
02400000	02FFFFFF	CE2	12M	Reserved
03000000	033FFFFF	CE3	4M	SDRAM (bank 1) or optional asynchronous expansion memory
03400000	03FFFFFF	CE3	12M	Reserved
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 ’C6x EVM 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 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 EVM 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 ’C6x EVM does not use the CLKOUT1 and CLKOUT2 outputs, so these clocks should be disabled to minimize EMI emissions. The SDCLK output (which is used to clock the EVM’s four SDRAM devices, the CPLD, and the daughterboard circuitry) should have its polarity identical to the DSP’s internal SDCLK. 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 133 MHz or less. When the CPU clock rate is greater than133 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 0x3060 is recommended. For a full-rate SSCLK operation, a value of 0x3064 is recommended.

The EMIF CE space control registers for the CE0–CE3 must be initialized to select the external memory configuration of the ’C6x EVM. *The CE1 memory space control register must be configured for a strobe period of 3 CLKOUT1 cycles for proper operation.* Table 1–5 summarizes the CE space allocation of the ’C6x EVM.

Table 1–5. TMS320C6x EVM 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, audio codec, and expansion memory	32-bit async, strobe = 3	0x1800004	OSC B×4: 0x40F40323 OSC A×4: 0x40F40323 OSC B: 0x10D10321 OSC A : 0x10D10321
CE2	SDRAM (bank 0)	100 MHz maximum	0x1800010	0x30
CE3	SDRAM (bank1)	100 MHz maximum	0x1800014	0x30

Because the ’C6x EVM 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 (OSC A or OSC B) and clock mode (multiply-by-1 or multiply-by-4). The SDRAM devices have a t_{RC} (refresh/active-to-refresh/active) period of 100 ns, a t_{RP} (precharge-to-active) period of 30 ns, and a t_{RCD} (active-to-read/write) period of 30 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 1–6.

Table 1–6. EMIF SDRAM Control Register Timing Fields

CPU Clock (MHz)	CLKOUT2 Period (ns)	TRC Field	TRP Field	TRCD Field
25	??	??	??	??
33	60	1	0	0
40	50	1	0	0
100	??	??	??	??
133	15	6	1	1
160	12.5	7	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 1–7.

Table 1–7. EMIF SDRAM Timing Register Values

CPU Clock (MHz)	CLKOUT2 Period (ns)	Maximum Register Value
OSC A	60	0x103
40	50	0x138
133	15	0x410
160	12.5	0x4E1

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

1.5 Expansion Interfaces

The ’C6x EVM provides two expansion connectors that allow a daughterboard to be connected to the board. Daughterboards can be used to extend the capabilities of the EVM 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 EVM board. The use of TI ’LVTH and ’CBT family interface devices allows the use of either 5- or 3.3-V devices to be used on the daughterboard.

The ’C6x EVM’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 J6 on the EVM. The expansion peripheral interface connector is J7. See Chapter 3, 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 EVM, including nonvolatile memory that can be used to boot the EVM 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 such as other codecs and communication devices to be added to the EVM 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 audio codec. 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 (XEXT_INT7) is included on the expansion peripheral interface to allow the daughterboard to interrupt the 'C6201/6701 to notify it of data transfers and other significant events. This interrupt is pulled down on the EVM, 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 peripheral 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 EVM 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 ’C6x EVM 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 EVM 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.

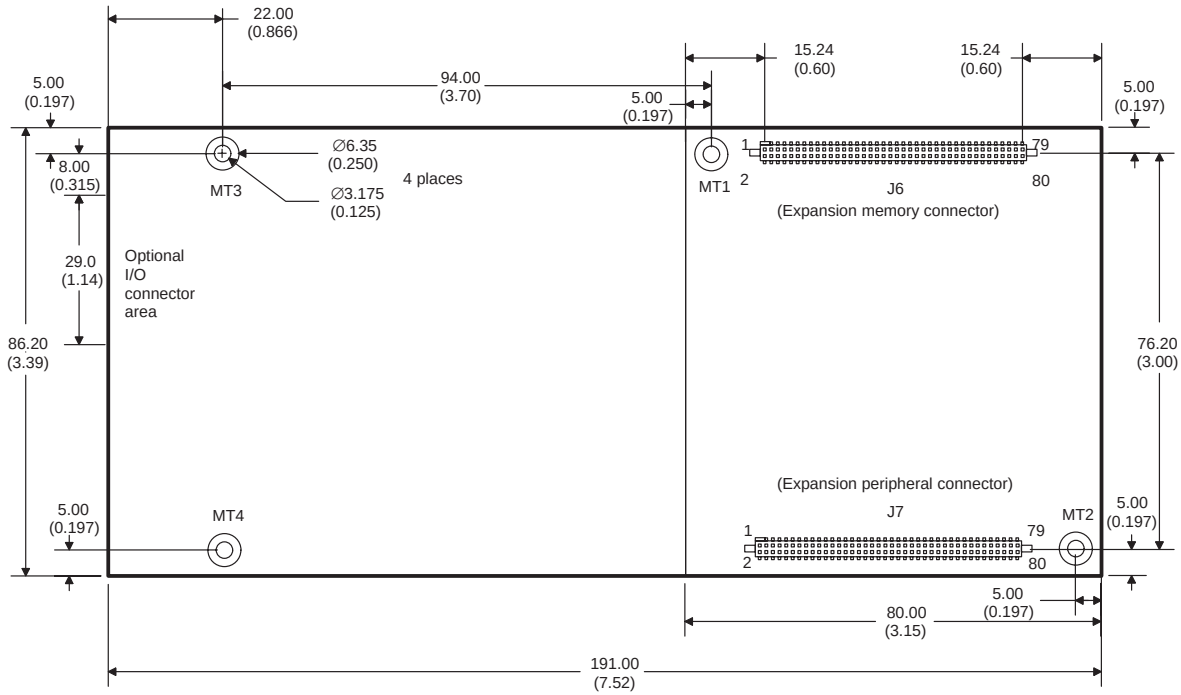
The large daughterboard measures approximately 7.5 inches × 3.4 inches and mounts from the center of the board over to the mating connector end of the board. This format is intended for daughterboards that require an I/O connection on the mounting bracket or need more space than the small daughterboard 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 EVM provides four standoff mounting holes to support daughterboard connections. Mounting holes MT1 and MT2 support small daughterboards, and all four mounting holes support large daughterboards. The MT1, MT2, and MT3 mounting holes are electrically connected to the digital ground plane to provide additional daughterboard grounding. MT4 is not electrically connected on the EVM.

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 ’C6x EVM board.

Figure 1–5. Daughterboard Envelopes and Connections on the TMS320C6x EVM



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 EVM 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 (MT1–MT4) on the EVM for standoff mounting.

9) Mounting holes MT1–MT3 are electrically connected to digital ground.

10) Mounting hole MT4 is floated with no electrical connection.

1.6 PCI Interface

The ’C6x EVM’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 EVM’s Windows drivers obtain information from the EVM’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 $\times$ 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 ’C6x EVM 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 EVM hardware from having to handle them directly. The ’C6x EVM’s 32-bit PCI interface operates at up to 33 MHz in a 5-V signaling environment. The EVM 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 ’C6x EVM is used to store PCI configuration information for the board including its vendor and device IDs, memory space requirements, and operational parameters. The EVM’s vendor ID is 0x104C (Texas Instruments), and its device ID is 0x1002.

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 EVM. This information is used by the PC BIOS for system resource allocation.

The EVM 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 ’C6x EVM 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), EVM control and status registers, and the ’C6201 HPI. Table 1–8 summarizes the ’C6x EVM PCI BAR definitions.

Table 1–8. TMS320C6x EVM 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	EVM 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 may 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 EVM is a PCI target for JTAG TBC and DSP HPI transactions.

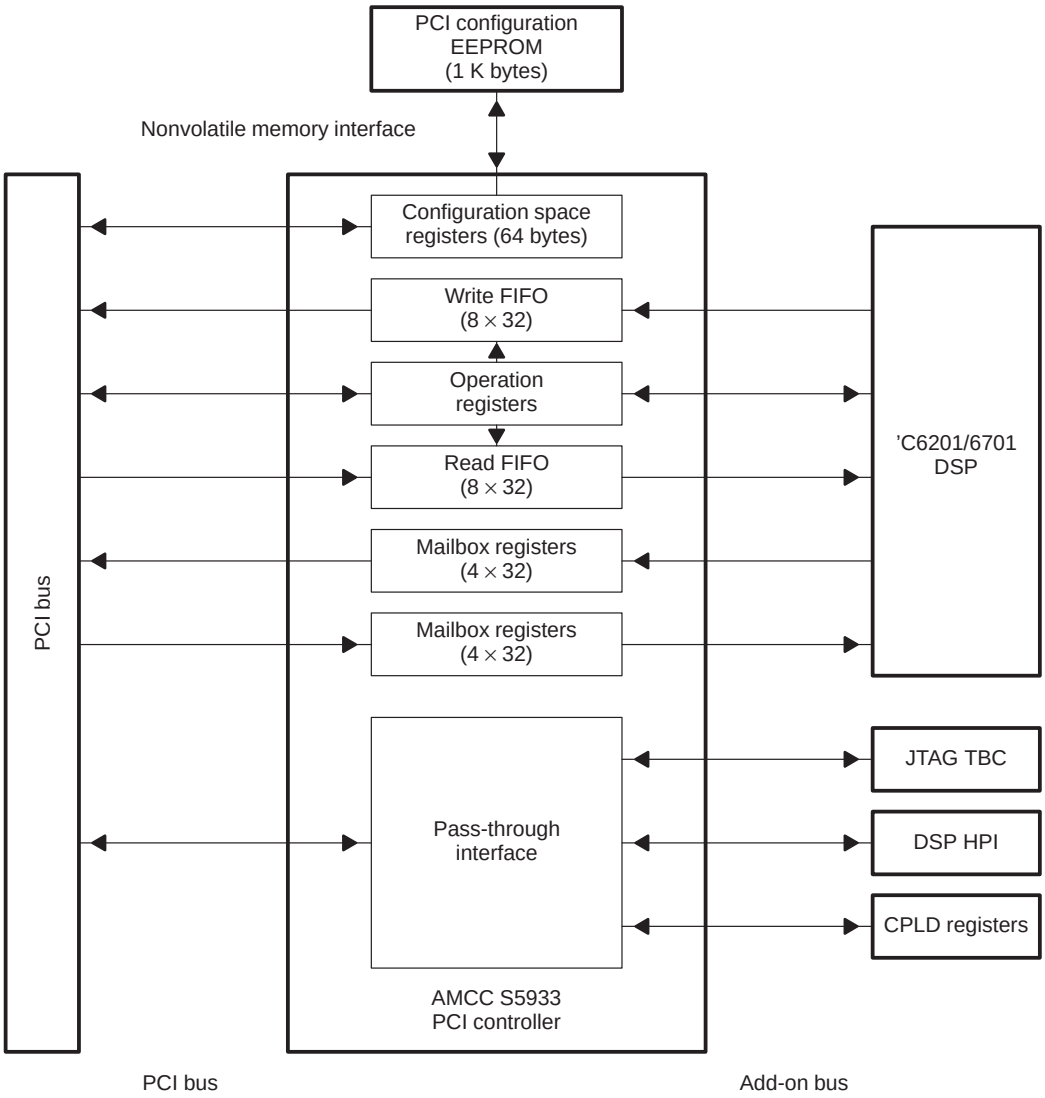
The S5933 registers and FIFOs are memory mapped into both the host and DSP memory spaces. The host can read the S5933 configuration registers to determine where the EVM is located in physical memory and what host interrupt has been assigned to it. The host software also configures the S5933 through the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

The EVM 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 ’C6x 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 ’C6x EVM’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 1–9 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 1–9. 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 ’C6201/6701 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 1–10 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 1–10. 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 ’C6x EVM’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 EVM’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 EVM’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 ’C6x EVM’s onboard JTAG TBC enables host software to control the ’C6201/6701 JTAG interface for testing and emulation purposes. The debugger uses this interface to control and monitor the DSP. The ’C6x EVM debugger shipped with the EVM 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 ’C6x EVM’s CPLD provides nine 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 ’C6x EVM 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, and check the revision number of the CPLD. 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 ’C6x EVM 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 ’C6x EVM supports both random and sequential accesses using the PCI controller’s BAR3 and BAR4 memory regions, respectively.

The three ’C6201/6701 HPI registers are memory mapped on the ’C6x EVM 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 1–11 summarizes the HPI registers mapped into BAR3.

Table 1–11. 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 ’C6x EVM handles HPI data transfers with the first half-word 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 ’C6x EVM 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 ’C6x EVM’s CPLD. The ’C6x EVM 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 ’C6x EVM 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 can be addressed as a linear array or a circular buffer by host software. 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 'C6x EVM'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 'C6201/6701 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 'C6201/6701 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 'C6x EVM supports bus mastering of the PCI bus, enabling the EVM to take control of the PCI bus and transfer data between the host and EVM 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 EVM 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.

Bus master transfers are supported by the S5933 using two on-chip FIFOs for read and write transfers between the 'C6201/6701 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 audio codec 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 'C6x EVM 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 ’C6x EVM 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 EVM 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-EVM and EVM-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 ’C6x EVM 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 to for the bus mastering to begin.

1.6.5.2 DSP Software Actions

The 'C6x EVM'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 'C6x EVM 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 ’C6x 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 ’C6x 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 ’C6x 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 ’C6x 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 EVM 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 ’C6201/6701 JTAG interface. This allows the ’C6x EVM debugger to be used with the EVM without an external emulator.

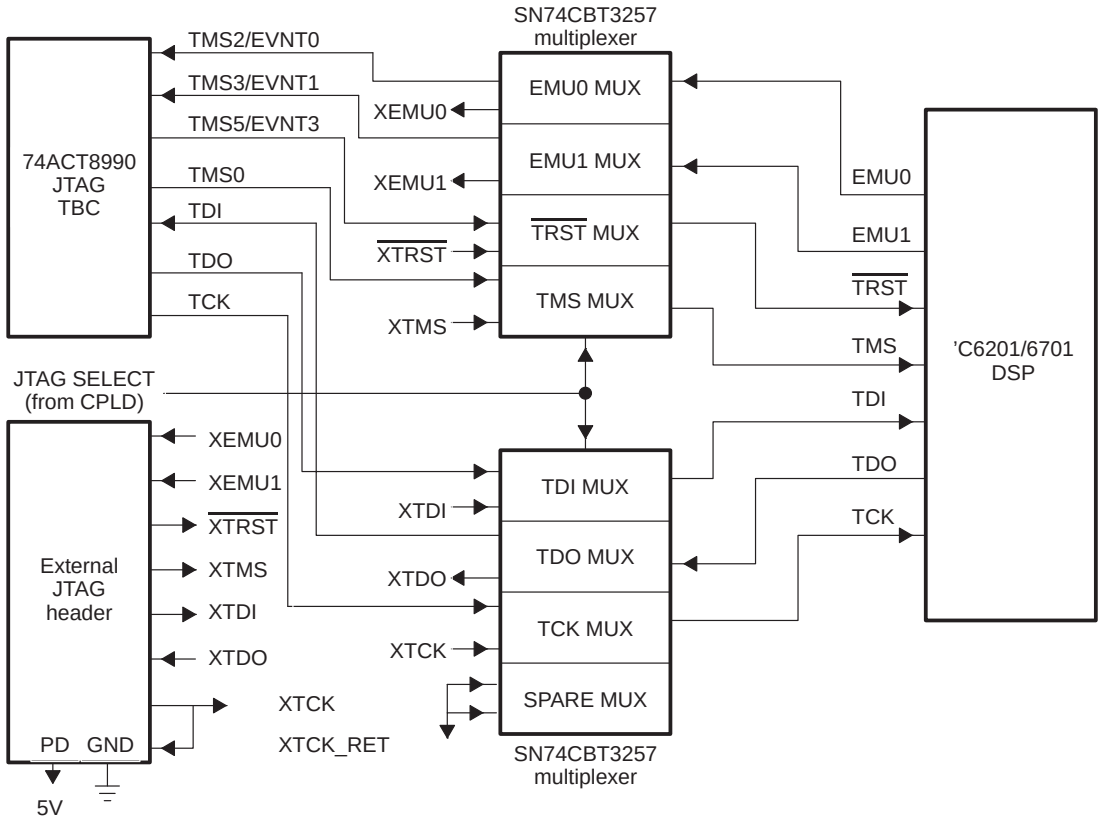
The ’C6x EVM’s embedded emulation support provides several benefits:

- 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 EVM (J5) supports an external XDS510 or XDS510WS emulator connection. This connection is required for debugging the EVM 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 $\overline{\text{TRST}}$ 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 are used to monitor the EMU0 and EMU1 signals from the 'C6201, respectively. The TMS5 signal is used to control the DSP's $\overline{\text{TRST}}$ 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 EVM provides a 16.625-MHz clock when the JTAG TBC is used for embedded emulation. The 16.625-MHz clock is derived from the EVM's OSC A-oscillator output using a single SN74F74 D-type flip-flop device configured for a divide-by-2 operation.

1.8 Programmable Logic

The 'C6x EVM 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:

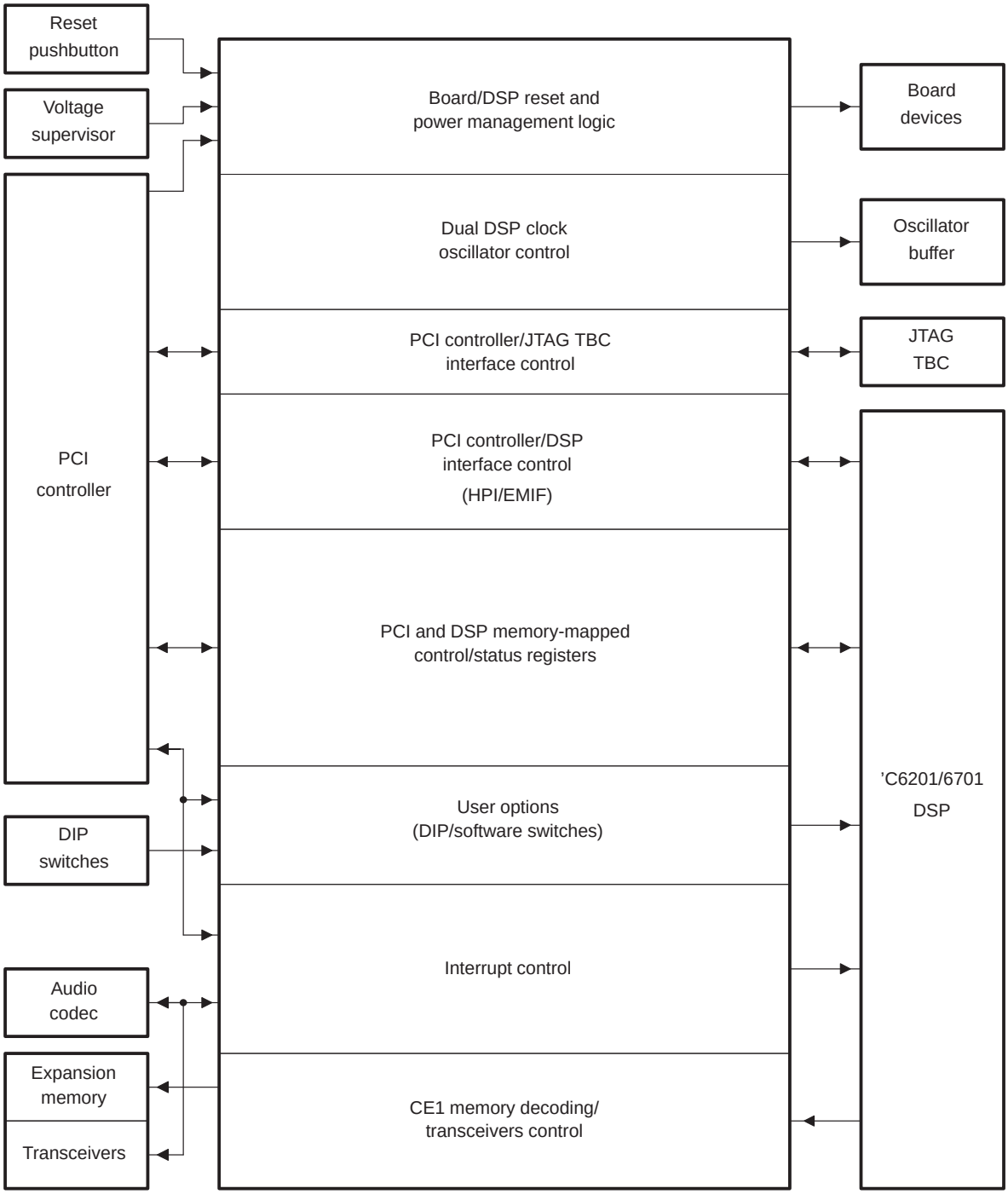
- Reset control
- Power management
- Dual DSP clock oscillator control
- PCI controller/DSP interface control
- PCI memory-mapped board control/status registers
- DSP memory-mapped control/status registers
- PCI and DSP interrupt control
- CE1 memory decoding
- Data transceivers control
- User options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

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 EVM. 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 EVM.

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

Figure 1–8. CPLD Interfaces and Functions



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 ’C6x EVM.

The EVM 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/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 EVM 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.

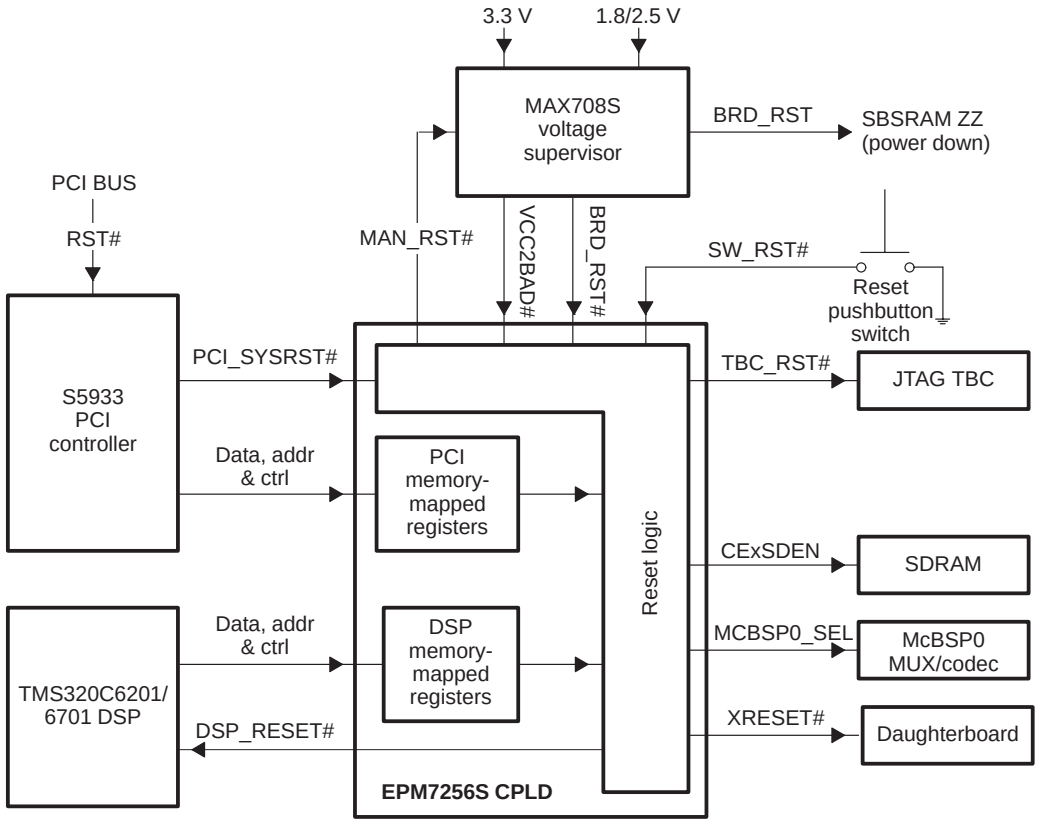
The board reset is produced when either the MAX708S indicates a reset (power up, undervoltage condition, or manual switch reset) or the PCI controller indicates a reset (PCI reset or manual software reset). The MAX708S provides two board reset signals: one active low and the other active high. The active-low board reset is routed to the CPLD, which directly controls the reset to the DSP, JTAG TBC, and daughterboard. The CPLD also indirectly controls the reset or power down of the audio codec via the McBSP0 multiplexer and the SDRAM via the clock enable control signals. 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 EVM 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 EVM 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 ’C6x EVM’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 ’C6x EVM CPLD includes logic that provides a power management capability that does not require additional devices. Various devices on the EVM can be placed in a low-power state to reduce power consumption.

The EVM’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 EVM 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 SDSRAM is disabled and put into a low-power mode by deasserting its CKE clock enable input signal. The McBSP0 multiplexer is configured to power down the audio codec by asserting the codec’s $\overline{\text{PWDN}}$ 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 1–12 lists the EVM 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 1–12. Power Management Device Control Summary

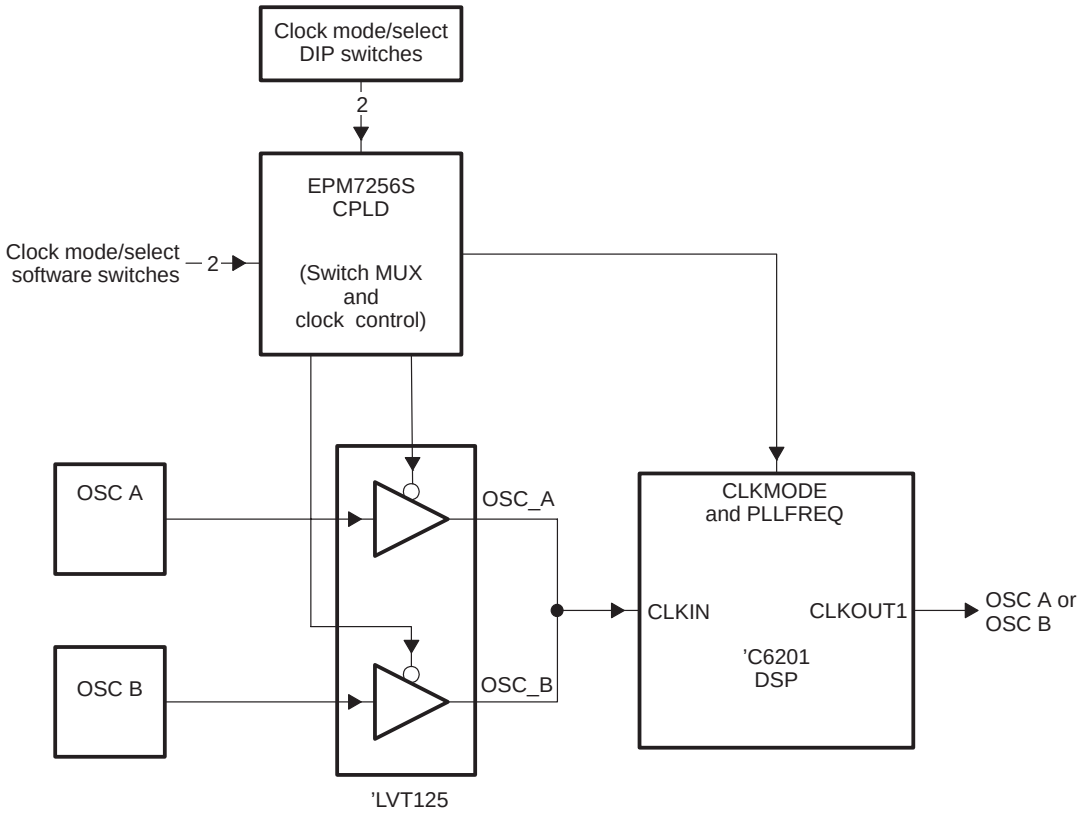
Device	Power Management Control Signals
DSP	$\overline{\text{RESET}}$ = 0
SBSRAM	ZZ = 1
SDRAM	CKE = 0
Audio codec	$\overline{\text{PDWN}}$ = 0
Daughterboard	XRESET = 0, DSP_PD = 1

1.8.3 Dual DSP Clock Oscillator Control

The ’C6x EVM includes dual DSP clock oscillator support to enable operation at the 160-MHz core clock rate with one-half-rate SBSRAM timing or at the full OSC A 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 EVM 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 EVM, 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 'LVTH162245 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 nine 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 EVM devices
- ❏ Read the user-option DIP switches
- ❏ Select the user options via software switches
- ❏ Read the selected DSP options
- ❏ Read the CPLD revision number

Table 1–13 summarizes the PCI memory-mapped CPLD registers.

Table 1–13. 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 –	CODECPD 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 –

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 1–14 summarizes the function of each bit in the CNTL register. Highlighted register values denote the power-up default values.

Table 1–14. 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, codec, DSP power down, daughterboard control, and user-defined LED status. Table 1–15 summarizes the function of each bit in the STAT register.

Table 1–15. 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)
5	CODECPD	R	Codec power-down status (0 = codec not in power down, 1 = codec in power down)
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 EVM'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 1–16 summarizes the function of each bit in the SWOPT register. Highlighted register values denote the power-up default values.

Table 1–16. 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 (**OSC A**, OSC B)
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 EVM'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 1–17 summarizes the function of each bit in the SWBOOT register. Highlighted register values denote the power-up default values.

Table 1–17. 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 EVM's onboard user option DIP switches. Table 1–18 summarizes the function of each bit in the DIPOPT register.

Table 1–18. 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 (OSC A , OSC B)
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 EVM's onboard boot mode DIP switches. Table 1–19 summarizes the function of each bit in the DIPBOOT register.

Table 1–19. 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 1–20 summarizes the function of each bit in the DSPOPT register.

Table 1–20. 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 (OSC A or OSC B)
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 1–21 summarizes the function of each bit in the DSPBOOT register.

Table 1–21. 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 EVM's CPLD. This information may be useful in assisting technical support. The revision number can be up to eight bits in length. Table 1–22 summarizes the function of each bit in the CPLDREV register.

Table 1–22. 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

1.8.7 DSP Memory-Mapped Control/Status Registers

The ’C6x EVM’s CPLD provides eight 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
- Select NMI source and enable
- Select McBSP0 connection/power-down codec
- 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

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 1–3 and Table 1–4. The eight registers are addressed sequentially on DWORD address boundaries.

The lower eight bits of the EMIF data bus are buffered by an ’LVTH162245 transceiver to provide the peripheral data bus (PDB[7:0]) that is connected to the CPLD.

Table 1–23 summarizes the eight DSP memory-mapped control and status registers implemented in the CPLD.

Table 1–23. DSP Memory-Mapped Control/Status CPLD Registers

Address MAP 1 (MAP 0)	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)	NMISEL RW 0 (host)	SP0SEL RW 0 (DB)	LED1 RW 0 (off)	LED0 RW 0 (off)
01780004 (01380004)	STAT	Status	XSTAT1 R –	XSTAT0 R –	DBINT R –	DSPNMI R –	CODECPD R –	CODECIRQ 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 –	ENDIAN 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 –
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)

The following subsections describe each of the eight 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 1–24 summarizes the function of each bit in the CNTL register. Highlighted register values denote the power-up default values.

Table 1–24. 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	NMISEL	RW	NMI interrupt select (**0 = PCI NMI register bit**, 1 = codec interrupt)
2	SP0SEL	RW	McBSP0 selection (**0 = daughterboard**, 1 = audio codec)
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 (0x1380004/0x1780004)

The STAT register enables the DSP software to monitor the daughterboard, interrupts, audio codec status, and PCI detection indicator. Table 1–25 summarizes the function of each bit in the STAT register.

Table 1–25. 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	CODECPD	R	Codec power-down status (0 = codec active, 1 = codec power down)
2	CODECIRQ	R	Codec interrupt status (0 = no interrupt, 1 = codec interrupt asserted)
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 (0x1380008/0x1780008)

The DIPOPT register enables the DSP software to read the DIP switch user options. Table 1–26 summarizes the function of each bit in the DIPOPT register.

Table 1–26. 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 (OSC A or OSC B)
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 (0x138000C/0x178000C)

The DIPBOOT register enables the DSP software to read the DIP switch boot mode selection. Table 1–27 summarizes the function of each bit in the DIPBOOT register.

Table 1–27. DSP 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_BMODE1	R	Switch boot mode 1
0	S_BMODE0	R	Switch boot mode 0

1.8.7.5 DSP DSPOPT Register (0x1380010/0x1780010)

The DSPOPT register enables the DSP software to read the actual DSP options. Table 1–28 summarizes the function of each bit in the DSPOPT register.

Table 1–28. 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 (OSC A or OSC B)
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 EVM is not installed in a PCI slot.

1.8.7.6 DSP DSPBOOT Register (0x1380014/0x1780014)

The DSPBOOT register enables the DSP software to read the DSP boot mode selection. Table 1–29 summarizes the function of each bit in the DSPBOOT register.

Table 1–29. 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
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 (0x1380018/0x1780018)

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 1–30 summarizes the function of each bit in the FIFOSTAT register. Highlighted register values denote the power-up default values.

Table 1–30. 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 (0x138001C/0x178001C)

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 1–31 summarizes the function of each bit in the SDCNTL register. Highlighted register values denote the power-up default values.

Table 1–31. 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**)

1.8.8 PCI and DSP Interrupt Control

The CPLD provides interrupt control that allows the host and DSP to interrupt each other and supports various EVM interrupts related to the daughterboard, PCI controller, audio codec, 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 DSP software can select whether the host or the audio codec sources NMI by setting the NMIEN and NMISEL bits in the DSP memory-mapped CNTL register provided by the CPLD.

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, codec, 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 ’C6x EVM are asserted on rising edges. Table 1–32 summarizes the DSP interrupts on the EVM.

Table 1–32. 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	Expansion peripheral interface (XEXT_INT7)
NMI	Host (PCI register) and audio codec (shared via CPLD logic)

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. They are generated by the CPLD in response to the PCI controller’s FIFO status. These interrupts are used to trigger the DSP’s DMA controllers or a CPU interrupt service routine.

EXT_INT7 is provided on the expansion peripheral interface (as XEXT_INT7) so that a daughterboard can interrupt the DSP to indicate significant events.

The NMI can be asserted by the stereo audio codec’s timer or by the host software via a memory-mapped control register on the PCI bus. The DSP software can select the NMI source by writing to the memory-mapped control register in the CPLD. The NMI is handled and driven by the CPLD. The codec’s timer can be disabled if it is not used. If enabled by the DSP’s software, it could be used as an additional timer or even a watchdog timer, providing flexibility.

1.8.9 CE1 Memory Decoding/Data Transceivers Control

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

The EVM uses ’LVTH162245 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 audio codec and the CPLD registers enable the peripheral bus transceiver. DSP EMIF accesses to the 3M-byte expansion memory space enable the expansion 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.

The CPLD provides audio codec chip select and read/write strobes when the audio codec memory space in CE1 is accessed.

A CPLD state machine, along with combinatorial logic, is used to generate the asynchronous ready (ARDY) signal to the DSP. Because the EVM 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 stand-alone. 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. A state machine is used to generate the ready indication when the audio codec and DSP memory-mapped registers in the CPLD are accessed based on a defined number of CLKOUT2 clock cycles. 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 1M $\times$ 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 EVM provides users 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 1–33 summarizes the EVM user options.

Table 1–33. 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 OSC A or OSC B 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 two onboard clock sources is used. The EVM provides four clock rates: OSC A and OSC A $\times$ 1, and OSC B and OSC B $\times$ 4.

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 'C6x EVM 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 EVM can operate in both PCI and stand-alone environments, it must support the selection of user options in both situations. When the EVM is operated outside the PC, DIP switches are used to configure the board. When the EVM 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 EVM uses its default switch settings. Therefore, in external operation, the DIP switches are used exclusively for user option selections.

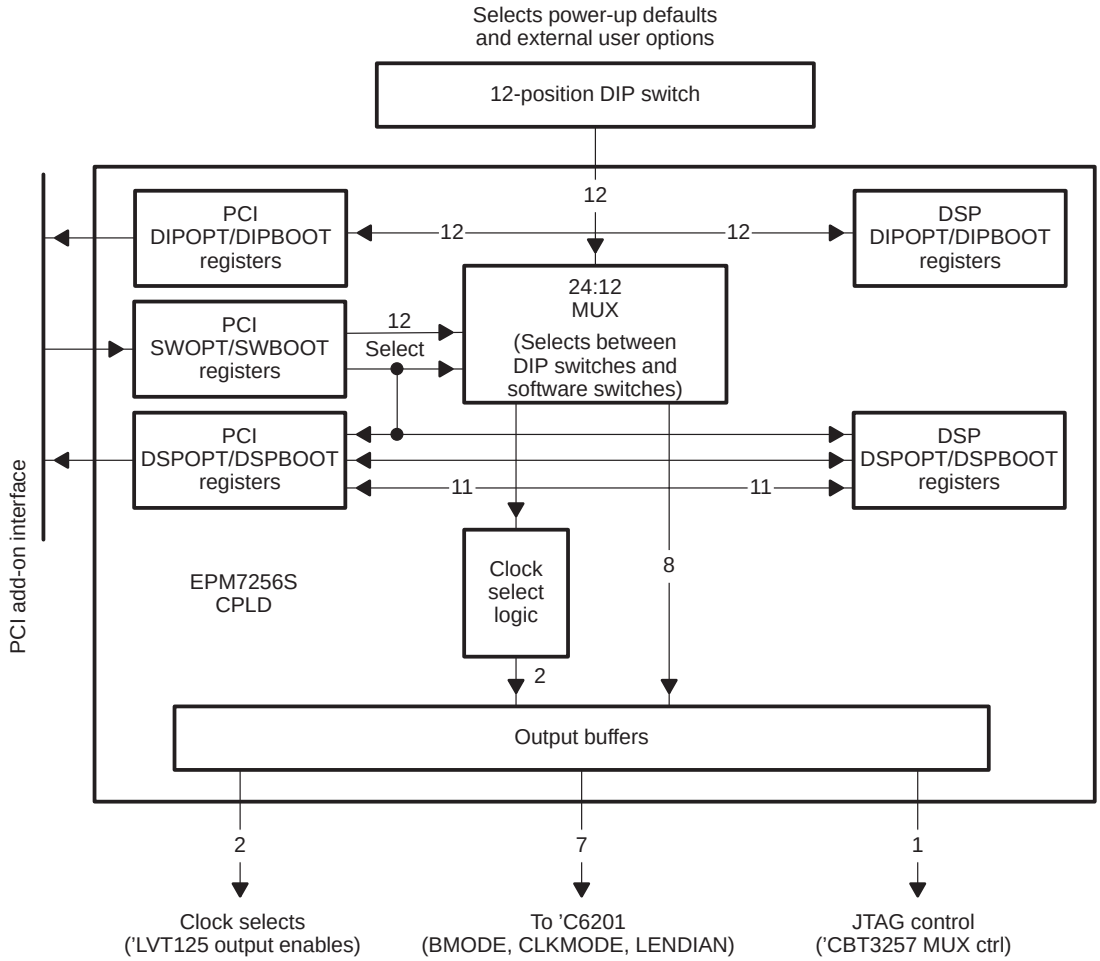
Figure 1–11 shows how this dual-use option support is implemented in the EVM's CPLD.

The EVM'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 EVM 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 EVM 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



1.9 Stereo Audio Interface

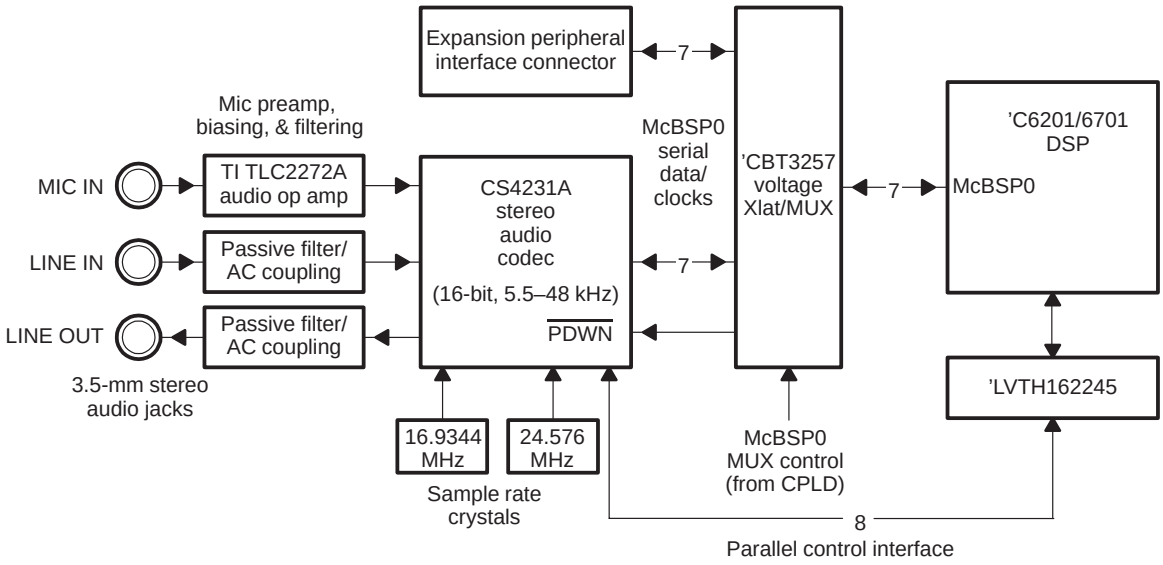
The ’C6x EVM includes a stereo audio interface that has the following characteristics:

- ❏ 16-bit stereo audio codec with sample rates from 5.5 kHz–48 kHz
- ❏ Linear (8- and 16-bit) and companded (μ-law and A-law) data formats
- ❏ Support for stereo electret microphones with voltage bias
- ❏ Stereo line-level input
- ❏ Stereo line-level output
- ❏ Input gain and output attenuation control
- ❏ Three 3.5-mm (1/8-inch) stereo audio jacks

Figure 1–12 shows the configuration of the ’C6x EVM’s stereo audio interface.

The CS4231A multimedia codec provides a flexible, general-purpose audio front-end for the EVM. The CS4231A is a 16-bit stereo device. It includes complete on-chip filtering plus analog mixing and programmable gain and attenuation. The device supports 16-bit linear, 8-bit linear, and 8-bit companded (μ-law and A-law) data formats.

Figure 1–12. TMS320C6x EVM Stereo Audio Interface



The CS4231A supports the following popular sample rates (in kHz):

5.5125	22.0500
6.6150	27.4286
8.0000	32.0000
9.6000	33.0750
11.025	37.8000
16.0000	44.1000
18.9000	48.0000

This wide range of sample rates is compatible with applications ranging from telecom to pro-audio. The codec uses two crystals with frequencies of 16.9344 MHz and 24.576 MHz as sample clock master sources.

The CS4231A includes a byte-wide parallel control/status interface, which is compatible with the ’C6201 asynchronous timing that is independent from its serial data interface to the DSP’s McBSP0. This independent parallel interface, which is memory-mapped into the ’C6201 asynchronous CE1 space, allows the DSP to configure and monitor the codec without impacting the serial data stream handler. This parallel interface can also be used for capture and playback data transfers from the DSP or even the host (via the HPI).

The CS4231A audio codec presents four control/status registers mapped into the DSP’s CE1 memory space starting at address 0x01320000 (MAP 0) or 0x01720000 (MAP 1). These four registers are all eight bits wide. They are addressed by the DSP on 32-bit word boundaries, and only the lower eight bits are valid. Table 1–34 summarizes the four audio codec registers.

Table 1–34. CS4231A Audio Codec Registers

DSP Address MAP 1 (MAP 0)	Audio Codec Register	Description	Access
01720000 (01320000)	R0	Index address register	Read/write
01720004 (01320004)	R1	Indexed data register	Read/write
01720008 (01320008)	R2	Status register	Read/write
0172000C (01320008)	R3	PIO data register	Read/write

The CS4231A has 32 other registers (I0–I32) that are indirectly accessed by using the index address register mapped at 0x1320000 or 0x01720000. The address of the indexed register is written to this register before writing or reading the data register at offset 1.

A benefit of the codec's control interface being memory-mapped in the DSP's CE1 memory space is that the codec can be directly controlled from the PC host software via the PCI-mapped HPI. This is useful in some user applications that want to control the codec. For example, a visual control panel application could present switches and knobs that directly control the operation of the codec in real time, including input gain, output attenuation, input selection, test loopbacks, and more.

The CS4231A presents a 5-V TTL-compatible interface that is connected to the DSP through low-cost, TI 'CBT3257 multiplexers on its serial interface, and a TI 'LVTH162245 transceiver that provides the EVM's peripheral data bus on its parallel interface, for voltage translation.

The CS4231A includes multiple inputs that support line and microphone levels. The line input and line output jacks can be used for connecting to a stereo source such as a compact disc or cassette tape player. The software has independent control over the left and right input selections and their gains. It also includes a line-level output with DSP software-controllable attenuation.

Three industry-standard 3.5-mm audio jacks are located on the board's mounting bracket. The tip of each jack is the left audio channel, and the ring is the right audio channel. Mono connections only use the left channel.

TI op amps are used for the microphone preamplifier and filtering. A voltage bias is provided to support the use of battery-powered and electret microphones. Passive filtering is included on the EVM between the codec and the audio jacks for increased performance. Ferrite beads and bypass capacitors are used on all audio interfaces to reduce noise coupling. A separate 5-V analog supply is provided by a TI voltage regulator to reduce noise on the audio signals.

The microphone input is designed for electret microphones that require a bias voltage. A dynamic microphone can be used if capacitors are used to block the bias voltage. The maximum allowable signal level to the microphone is 300 mV. The microphone interface provides an external gain of 10 dB and an additional user-selectable codec gain of 0 dB or 20 dB.

The line input signal level can be a maximum of 6 V_{pp} (2 V_{rms}). The line input signal is attenuated 6 dB before it is presented to the codec. Internal codec gain from 0 dB–22.5 dB can be selected. The line output signal level can be a maximum of 3 V_{pp} (1 V_{rms}). The line output signal can be muted and attenuated from 0 dB–94.5 dB. The line output cannot drive low-impedance headphones or speakers (i.e. 8 or 32 ohm) directly. The line output must be connected to 10K or greater loads. Table 1–35 summarizes the signal levels of the input and output audio interfaces.

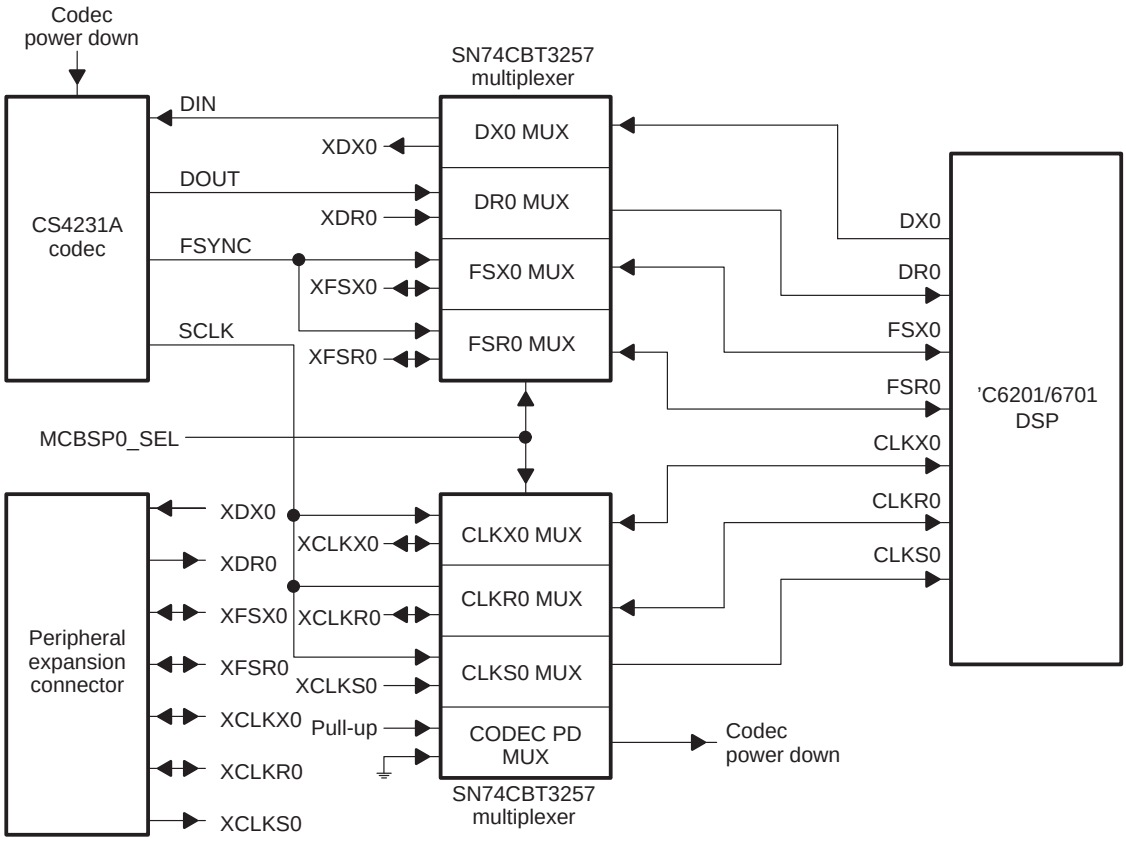
Table 1–35. Stereo Audio Interface Signal Levels

Audio Interface	Maximum Signal Level
Mic in with codec mic gain enabled (MGE = 1)	100 mV_{pp} (35 mV_{rms})
Mic in with codec mic gain disabled (MGE = 0)	1 V_{pp} (350 mV_{rms})
Line in	6 V_{pp} (2.1 V_{rms})
Line out	2.8 V_{pp} (1 V_{rms})

The CS4231A includes an on-chip 16-bit timer that can be useful to the DSP to supplement its two timers. The DSP can program a timer interval and the codec can interrupt the DSP, or the DSP can poll the timers counter register. The codec can be used as a watchdog timer because it is routed to the CPLD and can generate the DSP's nonmaskable interrupt (NMI).

The CS4231A serial interface is connected to the 'C6201 McBSP0 interface through a pair of TI 'CBT3257 quad 2:1 multiplexers as shown in Figure 1–13. This configuration allows the McBSP0 serial port to be connected to either the onboard audio codec or a daughterboard at any one time depending on your application. This allows daughterboards to use both of the DSP's serial ports, which is useful for many applications that do not require the audio codec. In addition to providing McBSP0 interface selection, one of the multiplexer outputs directly controls the codec's power-down input signal that reduces the codec's current from about 60 mA to 1 mA when the expansion interface is selected. The EVM defaults to the power-down state, and the audio codec's initialization library function controls a memory-mapped register bit in the CPLD's CNTL register to connect the codec to the DSP's McBSP0 interface. The CPLD CNTL register's SP0SEL bit must be set to 1 to activate the audio codec and allow it to accept configuration information. If the SP0SEL bit is 0, all accesses to the audio codec are ignored, and it always returns a value of 0x80.

Figure 1–13. McBSP0 Selection



1.10 Power Supplies

The EVM requires 1.8- or 2.5-V, 3.3-V, and 5-V supplies for its digital circuitry, as well as 5-V and 12-V supplies for its analog circuitry. An optional daughterboard –12-V supply is brought to the peripheral expansion interface from both the PCI bus and the external power connector. The digital 5 V and analog 12 V are obtained from the PCI bus during internal operation or an external power connector during external operation. The EVM's PCI connector's PRSNT1# and PRSNT2# pins are configured to indicate a maximum of 25 watts power dissipation. The EVM's other voltages are provided by onboard switching and linear voltage regulators that use the PCI or external 5-V and 12-V supplies as their inputs.

The PCI controller, JTAG TBC, serial EEPROM, oscillators, CPLD core, and 'CBT buffers/multiplexers require the digital 5 V from the PCI bus or external power connector. The audio codec and op amps require the analog 5 V, which is derived from the 12 V with a TI TL750L05 5-V regulator. The analog 5 V also has appropriate filtering and a separate, isolated analog ground plane to ensure a clean supply for the analog circuitry.

1.10.1 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 amps. 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 EVM 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.

1.10.2 1.8-V/2.5-V Voltage Regulator

The current revision 2.x DSP core requires 2.5 V for its core, and the future 0.18 micron version will require 1.8 V. The PT6502B ISR provides up to 8 A and is adjustable within a range of 1.3 V–2.6 V with external resistors. When a change is made to 0.18 micron silicon, the EVM can support the 1.8-V requirement with a simple resistor value change using the PT6502B, or a smaller, PT6407B 3A device can be used since the EVM provides a dual layout for both device packages. To support the large dynamic current requirements of the DSP, a 680-µF external capacitor is used at the voltage regulator's output, along with a carefully-designed bulk and decoupling capacitor arrangement.

1.10.3 5-V Voltage Regulator

The audio codec and the analog circuitry on the EVM require a clean, 5-V analog power supply for optimal performance. A TI TL750L05 low-dropout, linear voltage regulator that can provide up to 150 mA is used to provide the analog 5 V.

The TL750L05 requires an input voltage in the range of 6 V–26 V, so the PCI or external 12 V is used. The audio codec requires a maximum of 60 mA on the analog 5 V and the filter/buffer op-amps only require a few milliamps.

1.10.4 External Power Connector

The 'C6x EVM'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 EVM 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.

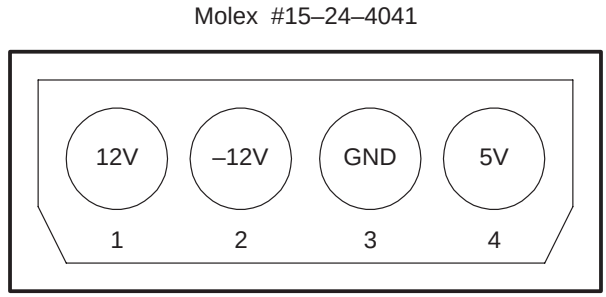
The user-supplied, external power supply should provide the following voltages and currents:

- ❑ 5 V_{DC} at 4A
- ❑ 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 'C6x EVM in a PCI slot.

Figure 1–14 shows the orientation of the external power connector pins.

Figure 1–14. External Power Connector



Note: Drawing is not to scale.

1.10.5 Fan Power Connector

The ’C6x EVM includes a small, 2-pin connector to supply 5 V (at 100 mA) to a low-profile, integrated heat sink/fan that attaches on top of the ’C6201 DSP. A passive heat sink is unable to dissipate enough heat in a worst case PC environment (50° C, 0 airflow) within the 0.57-inch height constraint of a PCI card. The integrated heat sink and fan reduces the DSP’s case temperature by 50–80%.

1.11 Voltage Supervision

The EVM'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- or 2.5-V power supplies, provides manual reset switch debounce, and generates a clean, 140-ms minimum reset pulse.

Because the 1.8- or 2.5-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 can be set via two external resistors configured as a voltage divider. Internally, the device triggers at 1.25 V. The resistor divider can set the external voltage trigger accordingly to support either 1.8 V or 2.5 V during manufacturing. The EVM is configured for a 2.3-V reset threshold for the 2.5-V voltage supply. An external resistor is also used to ensure a valid reset output down to 0 V.

The device also has a manual switch reset input that allows you to manually reset the DSP. This is important, particularly for the external EVM operation. The EVM'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, with one being active low and the other being 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 'C6x EVM.

Chapter 2

TMS320C6x EVM Host Support Software

This chapter describes the support software for the host that is provided with the EVM board. This software works on Pentium-based PCs running either Windows 95 or Windows NT 4.0. The software is installed during the EVM 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 ’C6x
- ❏ 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, *EVM 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 EVM board. By using the Win32 DLL, you can write user mode Win32 applications that execute under both operating systems.

These components, along with user mode host example code, are installed during the EVM software installation.

2.2 EVM Low-Level Windows Drivers

A low-level driver that is specific to the supported operating system handles all direct access to the one or more EVM boards in a system. For Windows 95, a Windows VxD, evm6x.vxd, is the low-level driver that provides access to the EVM hardware. For Windows NT 4.0, a kernel mode driver, evm6x.sys, provides the EVM hardware access.

All the functionality required for control and communication with the EVM 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 EVM Win32 DLL API

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

2.3.1 EVM 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 EVM board returns a board type of TYPE_EVM.

(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 EVM board. The enumerations that do not explicitly describe a 'C6x 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 EVM board. This type is used in the evm6x_send_message() and the evm6x_retrieve_message() functions.

2.3.2 EVM 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.

evm6x_abort_read

Terminate 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 one of the following values:

TRUE	Operation succeeded
FALSE	Operation failed

Example

In the following example, the evm6x_abort_read() function is used to abort 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

Terminate 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 one of the following values:

TRUE	Operation succeeded
FALSE	Operation failed

Example

In the following example, the evm6x_abort_write() function is used to abort 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

Retrieve Board Type and Version Information

Syntax

```
#include <evm6xdll.h>
BOOL evm6x_board_type(
        HANDLE                  h_device,
        PEVM6XDLL_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 *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 an EVM board is made, the location pointed to by the *p_board_type* parameter contains the enumerated value TYPE_EVM. 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 EVM board hardware revisions.

Return Value

The function returns one of the following values:

TRUE	Operation succeeded
FALSE	Operation failed

Example

In the following example, the evm6x_board_type() function is used to retrieve 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

Clear 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 one of the following values:

TRUE	Operation succeeded
FALSE	Operation failed

Example

In the following example, the evm6x_clear_message_event() function is used to clear the message event for an EVM 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 — *Close 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 one of the following values:

TRUE	Operation succeeded
FALSE	Operation failed

Example

In the following example, the evm6x_close() function is used to close a previously opened driver connection to an EVM 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

Display 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 one of the following values:

TRUE	Operation succeeded
FALSE	Operation failed

Example

In the following example, the evm6x_coff_display() function is used to display 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 — *Load 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 one of the following values:

TRUE	Operation succeeded
FALSE	Operation failed

Example

In the following example, the evm6x_coff_load() function is used to load 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

Return 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 of the CPLD registers and stores the values into the ULONG array you provide. The low byte of each ULONG is the only portion that is significant because 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 are filled with the CPLD register contents. This array must be at least 9 elements in size to hold all of the CPLD registers of the EVM board.

Return Value

The function returns one of the following values:

TRUE	Operation succeeded
FALSE	Operation failed

Example

In the following example, the evm6x_cpld_read_all() function reads the current state of the CPLD registers.

```
#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_generate_nmi_int

Generate 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 one of the following values:

TRUE	Operation succeeded
FALSE	Operation failed

Example

In the following example, the evm6x_generate_nmi_int() function is used to generate an NMI to the DSP on an EVM 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

Close 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 one of the following values:

TRUE	Operation succeeded
FALSE	Operation failed

Example

In the following example, the evm6x_hpi_close() function is used to close 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 — *Fill 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 one of the following values:

TRUE	Operation succeeded
FALSE	Operation failed

Example

In the following example, the evm6x_hpi_fill() function is used to fill 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

Generate 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 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 one of the following values:

TRUE	Operation succeeded
FALSE	Operation failed

Example

In the following example, the evm6x_hpi_generate_int() function is used to generate 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

Open 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 is used to open a handle to the HPI on an EVM 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 *Read 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 one of the following values:

TRUE	Operation succeeded
FALSE	Operation failed

Example

In the following example, the evm6x_hpi_read() function is used to read 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

Read a Single Byte, Short or Long, 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.

Note:

This call reads a 32-bit-aligned value from DSP memory and 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, do not use this call.

- 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 to 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 one of the following values:

TRUE	Operation succeeded
FALSE	Operation failed

Example

In the following example, the evm6x_hpi_read_single() function reads 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 — *Write 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 one of the following values:

TRUE	Operation succeeded
FALSE	Operation failed

Example

In the following example, the evm6x_hpi_write() function is used to write 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

Write 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.

Note:

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 one of the following values:

TRUE	Operation succeeded
FALSE	Operation failed

Example

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

```
#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 *Initialize 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 2-36). 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 one of the following values:

TRUE	Operation succeeded
FALSE	Operation failed

Example

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

Read 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 one of the following values:

TRUE Operation succeeded
FALSE Operation failed

Example

In the following example, the evm6x_nvram_read() function is used to read 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

Write 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.

> **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 one of the following values:

TRUE	Operation succeeded
FALSE	Operation failed

Example

In the following example, the evm6x_nvram_write() function is used to write 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 — *Open 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 is used to perform a non-exclusive 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 — *Read 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 2-42).

Also, a transfer can be terminated at any time using the evm6x_abort_read() function (see page 2-6). 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 one of the following values:

TRUE	Operation succeeded
FALSE	Operation failed

Example

In the following example, the evm6x_read() function is used to transfer 292 bytes (73 words, 32 bits each) from the DSP into host memory using the PCI FIFO interface provided on the EVM 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_time-out() function description on page 2-42 and the evm6x_abort_read() function description on page 2-6.

```
#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

Reset 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 2-32) 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 one of the following values:

TRUE	Operation succeeded
FALSE	Operation failed

Example

In the following example, the evm6x_reset_board() function is used to reset the second EVM 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 *Reset 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 2-43). 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 2-28).
3) Load the DSP program with HPI write calls using the evm6x_coff_load() function (see page 2-12).
4) Unreset the DSP from its halted state using the evm6x_unreset_dsp() function (see page 2-43).

Return Value

The function returns one of the following values:

TRUE	Operation succeeded
FALSE	Operation failed

Example

In the following example, the the evm6x_reset_dsp() function is used to place the DSP into HPI boot mode. While in this state, the DSP accepts HPI accesses. The evm6x_unreset_dsp() function is used to release the DSP from this state and to begin execution of DSP code. See the evm6x_unreset_dsp() function example on page 2-43 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

Retrieve a Message from a DSP

Syntax

```
#include <evm6xdll.h>
BOOL evm6x_retrieve_message(
        HANDLE                  h_device,
        PEVM6XDLL_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 one of the following values:

TRUE	Operation succeeded
FALSE	Operation failed

Example

In the following example, the evm6x_retrieve_message() function is used to check for and retrieve 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

Send a Message to a DSP

Syntax

```
#include <evm6xdll.h>
BOOL evm6x_send_message(
        HANDLE                  h_device,
        PEVM6XDLL_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 one of the following values:

TRUE	Operation succeeded
FALSE	Operation failed

Example

In the following example, the evm6x_send_message() function is used to send a 32-bit value, 0xfeed002f, to a DSP using the mailbox capability of the EVM 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

Set 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 2-36).

- 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 one of the following values:

TRUE	Operation succeeded
FALSE	Operation failed

Example

In the following example, the evm6x_set_board_config() function sets the EVM 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 — *Set 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 one of the following values:

TRUE	Operation succeeded
FALSE	Operation failed

Example

In the following example, the evm6x_set_timeout() function is used to set the bus master transfer time-out value to 5000 milliseconds (or 5 seconds). This means that any transfer that takes longer than 5 seconds 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

Unreset 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 2-36). 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 one of the following values:

TRUE	Operation succeeded
FALSE	Operation failed

Example

In the following example, the evm6x_unreset_dsp() function is used to release 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_write — *Write 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 2-42).

Also, a transfer can be terminated at any time using the evm6x_abort_write() 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 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 one of the following values:

TRUE	Operation succeeded
FALSE	Operation failed

Example

In the following example, the evm6x_write() function is used to send 292 bytes (73 words, 32 bits each) from host memory to the DSP using the PCI FIFO interface provided on the EVM 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 2-42 and the evm6x_abort_write() 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_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 EVM Host Support Software Example

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

Example 2–2. EVM 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

TMS320C6x EVM DSP Support Software

This chapter describes the EVM DSP support software by providing application programming interfaces (APIs) and example code for the multichannel buffered serial port (McBSP) driver, audio codec library, and board support library. All of these modules use the TMS320C6x 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 the McBSP driver, the codec library, and the board support library. The example code provided operates the CS4231A audio codec in serial data mode, and all audio data is communicated to and from the ’C6x via the McBSP port 0. Configuration and control of the codec is achieved via the parallel data interface. These two modules are totally independent of one another in that the codec 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 codec 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 ’C6x 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 codec library is a collection of routines that configure and control the operation of the CS4231A audio codec. The API functions correspond closely to the functional organization of the chip. Numerous macros are defined for this library in the file codec.h. These macros may be used as arguments to the API functions.

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

3.2 Using the DSP Support Software Components

The DSP support software, which consists of the McBSP driver, audio codec library, and board support library, is installed from the accompanying CD-ROM to the \evm6x\dsp\lib 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 'C6x 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 'C6x 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, audio codec library, and board support library exists in the .\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 'C6x 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 audio codec and 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 'C6x EVM debugger, or you can load the code and allow it to run using the EVM 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 multichannel buffered serial port (McBSP) driver API. Included in this discussion are the macros, data types, and defined functions that comprise the McBSP driver for the ’C6x EVM 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 common.h. The majority of the macros used by the McBSP driver are defined within the ’C6x 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	0	Flags McBSP as closed
OPEN	1	Flags McBSP as open and waiting for control
BUSY	2	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
{
  unsigned 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

Receive 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 is used to receive 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

Send 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 is used to send 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 — *Close 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 — *Configure 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 'C6x 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 'C6x 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 Send 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,
        Mcbsp_dev       frame_sync_dev,
        Callback        *p_wact);
```

Defined in

mcbspdrv.c as a callable C routine

Description

The mcbsp_cont_async_send() function is used to repeatedly send 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, then 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

Initialize 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 — *Open 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 — *Reset 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

None

Example

The code in this example resets the associated McBSP to its default state.

```
mcbsp_reset(dev);
```

mcbsp_stop — *Stop 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

Receive 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 is used to receive 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

Send 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 3-13).

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 Codec Library API

This section discusses the audio codec library API. Included in this discussion are the macros and defined functions that comprise the audio codec library for the ’C6x EVM board.

3.4.1 Codec Library Macros

Table 9–2 lists the macros defined, their values, and a description of each. The API functions that use each set of macros are also listed. These macros are defined within the public header file codec.h.

Table 3–2. Codec Library API Macros

(a) Mode selection macros

Macro	Value	Description
CAPTURE	1	Selects capture mode for control
PLAYBACK	2	Selects playback mode for control
BOTH	3	Applies arguments to both capture and playback modes

Note: These macros are used by the function codec_audio_data_format().

(b) Left/right channel selection macros

Macro	Value	Description
LEFT	1	Selects left channel for control
RIGHT	2	Selects right channel for control

Note: These macros are used by the following functions:
codec_adc_control()
codec_line_in_control()
codec_aux_control()
codec_dac_control()

(c) Auxiliary port selection macros

Macro	Value	Description
CHAN1	1	Selects auxiliary port 1 for control
CHAN2	2	Selects auxiliary port 2 for control

Note: These macros are used by the function codec_aux_control().

Table 3–2. Codec Library API Macros (Continued)

(d) Analog-to-digital converter (ADC) input gain macros

Macro	Value	Description
MIN_ADC_INPUT_GAIN	0.0	Minimum gain setting in dB for left and right input ADCs
MAX_ADC_INPUT_GAIN	22.5	Maximum gain setting in dB for left and right input ADCs
ADC_INPUT_GAIN_STEP	1.5	Gain setting resolution for left and right input ADCs

Note: These macros are used by the function codec_adc_control( ).

(e) Auxiliary and line mixer gain macros

Macro	Value	Description
MIN_AUX_LINE_GAIN	–34.5	Minimum gain setting in dB for AUX1, AUX2, and LINE mixer gain
MAX_AUX_LINE_GAIN	12.0	Maximum gain setting in dB for AUX1, AUX2, and LINE mixer gain
AUX_LINE_GAIN_STEP	1.5	Gain setting resolution for AUX1, AUX2, and LINE mixers

Note: These macros are used by the following functions:
codec_line_in_control()
codec_aux_control()

(f) Digital-to-analog converter (DAC) and loopback attenuation macros

Macro	Value	Description
MIN_DAC_LBACK_ATTEN	0.0	Minimum attenuation setting in dB for left and right DACs and left and right loopback attenuators
MIN_DAC_LBACK_ATTEN	94.5	Maximum attenuation setting in dB for left and right DACs and left and right loopback attenuators
DAC_LBACK_ATTEN_STEP	1.5	Attenuation setting resolution for DACs and loopback attenuators

Note: These macros are used by the following functions:
codec_dac_control()
codec_loopback_enable()

Table 3–2. Codec Library API Macros (Continued)

(g) ADC input selector macros

Macro	Value	Description
LINE_SEL	0	Selects LINE input as source for left or right ADC
AUX1_SEL	1	Selects AUX1 input as source for left or right ADC
MIC_SEL	2	Selects MIC input as source for left or right ADC
LINE_OUT_LBACK_SEL	3	Selects LINE OUT as source for left or right ADC

Note: These macros are used by the function codec_adc_control().

(h) Capture and playback audio data format macros

Macro	Value	Description
LINEAR_8BIT_UNSIGNED	0	Selects linear, 8-bit unsigned data format
ULAW_8BIT_COMPANDED	1	Selects μ-Law, 8-bit companded format
LINEAR_16BIT_SIGNED_LE	2	Selects linear, 16-bit signed little-endian data format
ALAW_8BIT_COMPANDED	3	Selects A-Law, 8-bit companded format
RSVD_FORMAT	4	Reserved data format designator
ADPCM_4BIT_IMA	5	Selects ADPCM, 4-bit, IMA-compatible data format
LINEAR_16BIT_SIGNED_BE	6	Selects linear, 16-bit signed big-endian data format

Note: These macros are used by the function codec_audio_data_format()

Table 3–2. Codec Library API Macros (Continued)

(i) Calibration mode macros

Macro	Value	Description
NO_CAL	0	No calibration is performed after resetting the MCE bit
CONVERTER_CAL	1	Calibrates only ADCs and DACs
DAC_CAL	2	Calibrates only the DAC interpolation filters
FULL_CAL	3	Calibrates all offsets, ADCs, DACs, and analog mixers

Note: These macros are used by the function codec_calibrate( ).

(j) Serial data format macros

Macro	Value	Description
SERIAL_64BIT_ENHANCED	0	Selects 64-bit enhanced serial audio data format; contains status information
SERIAL_64BIT	1	Selects 64-bit serial audio data format; does not contain status information
SERIAL_32BIT	2	Selects 32-bit serial audio data format; does not contain status information

Note: These macros are used by the function codec_serial_port_enable( ).

3.4.2 Codec Library API Functions

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

codec_adc_control

Control Input ADCs

Syntax

```
#include <codec.h>
int codec_adc_control(
        int     leftRightSel,
        float   adcGain,
        int     micGain,
        int     sourceSel );
```

Defined in

codec.c as a callable C routine

Description

The codec_adc_control() function controls the left and right input analog-to-digital converters (ADCs) in the CS4231A codec.

- The *leftRightSel* parameter selects between the left and right converters. Valid defined values are LEFT and RIGHT.
- The *adcGain* parameter specifies the ADC gain. Valid values are 0.0 dB–22.5 dB in .5-dB increments.
- The *micGain* parameter indicates whether or not to enable the 20-dB microphone input gains. Valid values are either TRUE (nonzero) or FALSE (zero).
- The *sourceSel* parameter selects the input source for the specified ADC. It can be one of these values:
  - LINE_SEL selects LINE input as the source.
  - AUX1_SEL selects AUX1 input as the source.
  - MIC_SEL selects MIC input as the source.
  - LINE_OUT_LBACK_SEL selects LINE OUT as the source.

Note:

The AUX1 input selection is provided for library completeness; however, the auxiliary inputs are not used on the EVM.

Return Value

The function returns one of the following values:

OK	Operation succeeded
ERROR	Operation failed

Example

This example sets the left ADC gain to 0 dB and disables the 20-dB microphone input gain. LINE is selected as the input source.

```
int status;
status= codec_adc_control( LEFT, 0.0, FALSE, LINE_SEL );
if (status == ERROR)
  return(ERROR);
```

codec_audio_data_format

Assign the Audio Data Format for Both Capture and Playback

Syntax

```
#include <codec.h>
int codec_audio_data_format(
        unsigned char   format,
        int             stereo,
        int             selection );
```

Defined in

codec.c as a callable C routine

Description

The codec_audio_data_format() function assigns the audio data format on the CS4231A codec.

- The *format* parameter selects the data format, which can be one of the following defined values:
  - ULAW_8BIT_COMPANDED selects μ-Law, 8-bit companded format.
  - LINEAR_16BIT_SIGNED_LE selects linear, 16-bit signed little-endian data format.
  - ALAW_8BIT_COMPANDED selects A-Law, 8-bit companded format.
  - ADPCM_4BIT_IMA selects ADPCM, 4-bit IMA-compatible format.
  - LINEAR_16BIT_SIGNED_BE selects linear, 16-bit signed big-endian data format.
  - LINEAR_8BIT_UNSIGNED selects linear, 8-bit unsigned data format.
- The *stereo* parameter selects the stereo format.
  - A value of TRUE (nonzero) selects stereo format, in which alternating samples represent left and right channels.
  - A value of FALSE (zero) selects mono format, in which the left channel sample occurs twice.
- The *selection* parameter selects the operating mode, which can be one of the following defined values:
  - CAPTURE selects capture mode.
  - PLAYBACK selects playback mode.
  - BOTH applies arguments to both the capture and the playback mode.

Return Value

The function returns one of the following values:

OK	Operation succeeded
ERROR	Operation failed

Example

This example selects the linear 16-bit signed little-endian data format operating in stereo mode for both capture and playback.

```
int status;
status = codec_audio_data_format( LINEAR_16BIT_SIGNED_LE,
                                  TRUE, BOTH);
if (status == ERROR)
 return(ERROR);
```

codec_aux_control

Control the Two Pairs of Auxiliary Inputs

Syntax

```
#include <codec.h>
int codec_aux_control(
        int     leftRightSel,
        int     channelSel,
        float   mixGain,
        int     mute );
```

Defined in

codec.c as a callable C routine

Description

This codec_aux_control() function controls the two pairs of auxiliary inputs on the CS4231A codec.

- The *leftRightSel* parameter selects between the left and right side inputs. Valid defined values are LEFT and RIGHT.
- The *channelSel* parameter selects between auxiliary port 1 and auxiliary port 2. Valid defined values are CHAN1 and CHAN2.
- The *mixGain* parameter specifies the auxiliary input mixer gain setting. Valid values are –34.5 dB–12.0 dB in 1.5-dB increments.
- The *mute* parameter, if TRUE (nonzero), mutes the selected line to the mixer.

Note:

This function is provided for library completeness; however, the auxiliary inputs are not used on the EVM.

Return Value

The function returns one of the following values:

OK	Operation succeeded
ERROR	Operation failed

Example

This example enables the left auxiliary 1 input mixer and sets its gain to 0 dB.

```
int status;
status = codec_aux_control( LEFT, CHAN1, 0.0, FALSE );
if (status == ERROR)
  return(ERROR);
```

codec_calibrate — *Calibrate the CS4231A Codec*

Syntax

#include <codec.h>
int **codec_calibrate(** int *calVal* **);**

Defined in

codec.c as a callable C routine

Description

The codec_calibrate() function calibrates the CS4231A codec.

The *calVal* parameter determines the calibration mode, which can be one of the following defined values:

- ❏ CONVERTER_CAL calibrates only the ADCs and DACs.
- ❏ DAC_CAL calibrates only the DAC interpolation filters.
- ❏ FULL_CAL calibrates all offsets, ADCs, DACs, and analog mixers.
- ❏ NO_CAL specifies that no calibration is performed after resetting the MCE bit.

Return Value

The function returns one of the following values:

OK	Operation succeeded
ERROR	Operation failed

Example

This example performs a full calibration on the CS4231A codec, which means the function calibrates all offsets, ADCs, DACs, and analog mixers.

```
int status;

status = codec_calibrate( FULL_CAL );
if (status == ERROR)
 return(ERROR);
```

codec_capture_disable — *Disable Capture Mode*

Syntax

#include <codec.h>
void **codec_capture_disable(** void **);**

Defined in

codec.c as a callable C routine

Description

The codec_capture_disable() function disables capture mode on the CS4231A codec.

Return Value

None

codec_capture_enable

Enable Capture Mode

Syntax

#include <codec.h>
void **codec_capture_enable(** void **);**

Defined in

codec.c as a callable C routine

Description

The codec_capture_enable() function enables capture mode on the CS4231A codec.

Return Value

None

codec_change_sample_rate

Set Capture and Playback Sample Rates on the CS4231A Codec

Syntax

#include <codec.h>
int **codec_change_sample_rate(**
int *sampleRate*,
int *returnFromMce* **);**

Defined in

codec.c as a callable C routine

Description

The codec_change_sample_rate() function changes the capture and playback sample rates on the CS4231A codec.

- The *sampleRate* parameter sets the sample rate in Hz. The given *sampleRate* is matched to a table of valid values, and the actual sample rate is set to the closest value found.
- The *returnFromMce* parameter, if TRUE, causes this function to clear the MCE bit before returning to the caller. If FALSE, the function returns to the caller with the codec in MCE mode.

Return Value

The function returns the actual sample rate (in Hz).

Example

This example sets the codec sample rate to the value closest to 44 kHz.

```
int actualSampleRate;
actualSampleRate= codec_change_sample_rate(44000,TRUE);
```

codec_dac_control

Control the Output DACs

Syntax

```
#include <codec.h>
int codec_dac_control(
        int     leftRightSel,
        float   dacAtten,
        int     mute );
```

Defined in

codec.c as a callable C routine

Description

The codec_dac_control() function controls the left and right output digital-to-analog converters (DACs) on the CS4231A codec.

- The *leftRightSel* parameter selects between the left and right DACs. Valid defined values are LEFT and RIGHT.
- The *dacAtten* parameter specifies the attenuation applied to the output of the selected DAC. Valid values are 0.0 dB–94.5 dB in 1.5-dB increments.
- The *mute* parameter, if TRUE (nonzero), mutes the selected line to the mixer.

Return Value

The function returns one of the following values:

OK	Operation succeeded
ERROR	Operation failed

Example

This example enables the right output DAC and sets its attenuation to 3 dB.

```
int status;

status = codec_dac_control( LEFT, 3.0, FALSE );
if (status == ERROR)
  return(ERROR);
```

codec_id — *Return Codec ID, Codec Version, and Chip ID*

Syntax

```
#include <codec.h>
int codec_id(
        int *pVersion,
        int *pChipId );
```

Defined in

codec.c as a callable C routine

Description

The codec_id() function returns the codec ID, codec version, and chip ID.

- The *pVersion* parameter is a pointer to the caller-declared integer in which to place the chip version.
- The *pChipId* parameter is a pointer to the caller-declared integer in which to place the chip ID.

Return Value

The function returns the codec ID.

Example

The following example returns the codec ID.

```
int version;
int chipId;
int codec_id;

codec_id  = codec_id( &version, &chipId);
```

codec_init — *Perform Required Codec Initialization and Default Configuration*

Syntax

```
#include <codec.h>
int codec_init( void );
```

Defined in

codec.c as a callable C routine

Description

The codec_init() function initializes the CS4231A codec, sets default parameters for the codec, and calibrates it.

Return Value

The function returns one of the following values:

OK	Operation succeeded
ERROR	Operation failed

Example

The following example initializes the codec for use.

```
int status;

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

codec_interrupt_disable

Disable Interrupt Mode

Syntax

#include <codec.h>
void **codec_interrupt_disable(** void **)**;

Defined in

codec.c as a callable C routine

Description

The codec_interrupt_disable() function disables interrupts from the CS4231A codec.

Return Value

None

codec_interrupt_enable

Enable Interrupts from CS4231A Codec

Syntax

#include <codec.h>
void **codec_interrupt_enable(** void **);**

Defined in

codec.c as a callable C routine

Description

The codec_interrupt_enable() function enables interrupts from the CS4231A codec.

Return Value

None

codec_line_in_control

Control Left and Right Line Inputs

Syntax

#include <codec.h>
int **codec_line_in_control(**
int *leftRightSel*,
float *mixGain*,
int *mute* **);**

Defined in

codec.c as a callable C routine

Description

The codec_line_in_control() function controls the left and right line inputs on the CS4231A codec.

- The *leftRightSel* parameter selects between the left and right line inputs. Valid defined values are LEFT and RIGHT.
- The *mixGain* parameter specifies the line input mixer gain setting. Valid values are –34.5 dB–12.0 dB in 1.5-dB increments.
- The *mute* parameter, if TRUE (nonzero), mutes the selected line to the mixer.

Return Value The function returns one of the following values:

OK	Operation succeeded
ERROR	Operation failed

Example This example enables the right line input mixer and sets its gain to 0 dB.

```
int status;

status = codec_line_in_control( RIGHT, 0.0, FALSE);
if (status == ERROR)
  return(ERROR);
```

codec_loopback_disable

Disable Loopback Mode

Syntax

#include <codec.h>
void **codec_loopback_disable(** void **);**

Defined in codec.c as a callable C routine

Description The codec_loopback_disable() function disables loopback mode on the CS4231A codec.

Return Value None

codec_loopback_enable

Enable Loopback Mode

Syntax

#include <codec.h>
int **codec_loopback_enable(** float *loopbackAtten* **);**

Defined in codec.c as a callable C routine

Description The codec_loopback_enable() function enables loopback mode on the CS4231A codec and sets loopback attenuation.

- The *loopbackAtten* parameter selects the loopback attenuation. Valid values are 0.0 dB–94.5 dB in 1.5-dB increments.

Return Value The function returns one of the following values:

OK	Operation succeeded
ERROR	Operation failed

codec_playback_disable

Disable Playback Mode

Syntax

#include <codec.h>
void **codec_playback_disable(** void **);**

Defined in

codec.c as a callable C routine

Description

The codec_playback_disable() function disables playback mode on the CS4231A codec.

Return Value

None

codec_playback_enable

Enable Playback Mode

Syntax

#include <codec.h>
void **codec_playback_enable(** void **);**

Defined in

codec.c as a callable C routine

Description

The codec_playback_enable() function enables playback mode on the CS4231A codec.

Return Value

None

codec_reset

Reset Codec

Syntax

#include <codec.h>
void **codec_reset(** void **);**

Defined in

codec.c as a callable C routine

Description

The codec_reset() function resets the CS4231A codec by asserting then de-asserting the PWDN pin. All registers are configured to their POR values.

Return Value

None

codec_serial_port_disable

Disable Serial Port Transfers on CS4231A Codec

Syntax

#include <codec.h>
int **codec_serial_port_disable(** void **);**

Defined in

codec.c as a callable C routine

Description

The codec_serial_port_disable() function disables the serial port on the CS4231A codec.

Return Value

The function returns one of the following values:

OK	Operation succeeded
ERROR	Operation failed

codec_serial_port_enable

Enable Serial Port Transfers on CS4231A Codec

Syntax

#include <codec.h>
int **codec_serial_port_enable(** int *serialFormat* **);**

Defined in

codec.c as a callable C routine

Description

The codec_serial_port_enable() function enables the serial port on the CS4231A codec in the selected format.

- The *serialFormat* parameter selects the serial data format for the CS4231A codec. Valid defined values are as follows:
  - SERIAL_64BIT_ENHANCED selects 64-bit serial audio data format enhanced with status information.
  - SERIAL_64BIT selects 64-bit serial audio data format with no status information.
  - SERIAL 32BIT selects 32-bit serial audio data format with no status information.

Return Value

The function returns one of the following values:

OK	Operation succeeded
ERROR	Operation failed

codec_set_base — *Set Base Codec Address Based on Memory Map and Endian Mode*

Syntax

```
#include <codec.h>
void codec_set_base(
        int   mapMode,
        int   littleEndian );
```

Defined in

codec.c as a callable C routine

Description

The codec_set_base() function configures the codec library for use based on the map and endian mode selected. This function is called from evm_init() and therefore does not need to be called directly (see page 3-44 for a description of the evm_init() function).

- The *mapMode* parameter indicates the current 'C6x memory map, with valid values of 0 or 1.
- The *littleEndian* parameter is TRUE if the 'C6x is in little-endian mode; otherwise, it is FALSE.

Return Value

None

codec_timer — *Use Internal Codec Timer*

Syntax

```
#include <codec.h>
int codec_timer(
        int usec,
        void (*fp)(void) );
```

Defined in

codec.c as a callable C routine

Description

The codec_timer() function controls the internal CS4231A timer in either a polled (blocking) or interrupt driven (nonblocking) fashion.

- The *usec* parameter specifies the number of microseconds to wait.
- If *fp* is NULL, this function returns after the specified number of microseconds. If *fp* is not NULL, this function returns immediately and the interrupt service routine pointed to by *fp* is called after the specified number of microseconds.

> **Note:**
>
> The maximum time supported by this routine is 650 000 µs (0.650 s). This is a limitation of the codec timer's 10-µs clock period and 16-bit counter register.

Return Value

The function returns one of the following values:

OK	Operation succeeded
ERROR	Operation failed

Example

This example demonstrates using the codec timer in interrupt-driven mode. Global variable timerExpired indicates to the main program that the codec timer has expired and has generated an interrupt.

```
interrupt void codec_timerISR( void )
{
  timerExpired = TRUE;
  return;
}

int timerExpired;
int status;

timerExpired = FALSE;
status= codec_timer( 100, codec_timerISR );
if (status == ERROR)
  return(ERROR);
```

3.5 Board Support Library API

This section discusses the EVM board support library API. Included in this discussion are the functions defined that comprise the board library for the ’C6x EVM 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.

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 ’C6x 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 CS4231A codec from the ’C6x McBSP port 0, placing it in power-down mode. This function connects McBSP port 0 to the daughterboard expansion connector.

Return Value

The function returns one of the following values:

OK	Operation succeeded
ERROR	Operation failed

evm_codec_enable

Connect Codec to ’C6x 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 connects the CS4231A codec to the ’C6x McBSP port 0, removing it from power-down mode. This function disconnects McBSP port 0 from the daughterboard expansion connector.

Return Value

The function returns one of the following values:

OK	Operation succeeded
ERROR	Operation failed

evm_db_reset

Hold Daughterboard in Reset

Syntax

```
#include <board.h>
void evm_db_reset(void);
```

Defined in

board.c as a callable C routine

Description

The evm_db_reset() function enables the reset signal to the daughterboard connector. Use the evm_db_unreset() function to disable the reset signal.

Return Value

None

evm_db_unreset — *Remove Daughterboard Reset*

Syntax

#include <board.h>
void **evm_db_unreset(**void**);**

Defined in

board.c as a callable C routine

Description

The evm_db_unreset( ) function disables the reset signal to the daughterboard connector. Use the evm_db_reset( ) function to enable the reset signal.

Return Value

None

evm_default_emif_init — *Initialize EMIF for EVM 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 EVM 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 EVM 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 EVM board for use by configuring base address variables based upon the 'C6x 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 EVM board for use.

```
int  status;

status = evm_init( );
if (status == ERROR)
  return(ERROR);
```

evm_led_disable — *Disable Selected EVM 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 EVM 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 EVM 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 EVM 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 ’C6x by clearing the NMIEN bit in the CNTL register of the EVM 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 ’C6x by setting the NMIEN bit in the CNTL register of the EVM CPLD.

Return Value

None

evm_nmi_sel — *Select 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 the external NMI to the ’C6x on the EVM board.

- If *sel* is 0, the host is selected as the source for the NMI; otherwise, the codec is selected as the source for the NMI.

Return Value

None

3.6 DSP Support Software Examples

The following module provides examples of configuring and controlling the McBSP and audio codec. The board support library is also used in the examples.

Example 3–2. McBSP, Audio Codec, and Board Support Sample Code

(a) codec_ex.c file

```
/*************************************************************************/
/*  <CODEC_EX.C> – Example code for CODEC and McBSP use.                  */
/*                                                                        */
/*     This module provides examples using the CS4231A audio CODEC.      */
/*     The data interface to the CS4231A on the EVM is the McBSP.        */
/*                                                                        */
/*  FUNCTIONS:                                                            */
/*    CodecExLb()         –  CODEC Example: Loopback mode                 */
/*    CodecExBlkCapPb()   –  CODEC Example: Block Capture and Playback    */
/*    CodecExContTonePb() –  CODEC Example: Continuous Tone Playback      */
/*    CodecExContCapPb()  –  CODEC Example: Continuous Capture and Playback */
/*                                                                        */
/*  STATIC FUNCTIONS:                                                     */
/*    ContTonePbCallback() – Callback function.                           */
/*    GenTone()            – Synthesize tone of specified frequency       */
/*                                                                        */
/*  GLOBAL VARIABLES DEFINED                                              */
/*    None.                                                               */
/*                                                                        */
/*************************************************************************/
/*-----------------------------------------------------------------------*/
/* INCLUDES AND LOCAL DEFINES                                            */
/*-----------------------------------------------------------------------*/
#include <stdio.h>
#include <stdlib.h>
#include <mcbsp.h>      /* mcbsp devlib                                  */
#include <common.h>
#include <mcbspdrv.h>   /* mcbsp driver                                  */
#include <board.h>      /* EVM library                                   */
#include <codec.h>      /* codec library                                 */
#include <mathf.h>
#include ″codec_ex.h″
#define VOICE_BAND_SAMPLE_RATE     8000 /* Hz  */
#define PRO_AUDIO_SAMPLE_RATE     44100 /* Hz  */
#define TONE_PB_SAMPLE_RATE        5512 /* Hz  */
#define BLOCK_CAPTURE_TIME            4 /* sec */
#define CONT_CAPTURE_TIME             1 /* sec */
#define CONT_BUFF_SIZE          0x3fffc /* 1 64kword frames per block       */
#define TONE_FREQ                  1000 /* Hz                               */
#define CONT_TONE_PB_SECS            30 /* secs                             */
#define NUM_BUFFS_TO_PLAY            20
#define PRINT_DBG 0
```

```
/*-----------------------------------------------------------------------*/
/* GLOBAL VARIABLES                                                      */
/*-----------------------------------------------------------------------*/
volatile int numBuffsPlayed;
/*-----------------------------------------------------------------------*/
/* FILE LOCAL (STATIC) VARIABLES                                         */
/*-----------------------------------------------------------------------*/
int asyncBuffCnt;
/*-----------------------------------------------------------------------*/
/* FILE LOCAL (STATIC) PROTOTYPES                                        */
/*-----------------------------------------------------------------------*/
void ContTonePbCallback(Mcbsp_dev dev,int status);
int GenTone(                          /* RET: OK or ERROR                */
            short         *pBuffer,    /* I/O: buffer for generated data    */
            int           buffSize,    /*  IN: number of shorts in buffer   */
            float        toneFreq,    /*  IN: requested tone frequency (Hz) */
            float        sampleRate,  /*  IN: current CODEC sample rate (Hz)*/
            int          leftRightSel);
void AsyncTxCallback(Mcbsp_dev dev,int status);
/*-----------------------------------------------------------------------*/
/* FUNCTIONS                                                             */
/*-----------------------------------------------------------------------*/
/*************************************************************************/
/* CodecExLb –  CODEC Example: Loopback mode                             */
/*                                                                       */
/*   This function provides an example of using the CS4321A CODEC in     */
/*   Digital Loopback Mode (DLB).                                        */
/*                                                                       */
/*************************************************************************/
int CodecExLb(                  /* RET: OK or ERROR                      */
         int micSel             /* IN : TRUE – mic sel, FALSE – line sel   */
             )
{
  if (codec_init())
    return(ERROR);
  if (micSel)
  {
    /* A/D 0.0 dB gain, turn on 20dB mic gain, sel (L/R)MIC as inputs       */
    if (codec_adc_control(LEFT,0.0,TRUE,MIC_SEL))
      return(ERROR);
    if (codec_adc_control(RIGHT,0.0,TRUE,MIC_SEL))
      return(ERROR);
  }
  else
  {
    /* A/D 0.0 dB gain, turn off 20dB mic gain, sel (L/R)LINE input         */
    if (codec_adc_control(LEFT,0.0,FALSE,LINE_SEL))
      return(ERROR);
    if (codec_adc_control(RIGHT,0.0,FALSE,LINE_SEL))
      return(ERROR);
    /* mute (L/R)LINE input to mixer                                       */
    if (codec_line_in_control(LEFT,MIN_AUX_LINE_GAIN,TRUE))
      return(ERROR);
```

```
    if (codec_line_in_control(RIGHT,MIN_AUX_LINE_GAIN,TRUE))
      return(ERROR);
  }
  /* D/A 0.0 dB atten, do not mute DAC outputs                              */
  if (codec_dac_control(LEFT, 0.0, FALSE))
    return(ERROR);
  if (codec_dac_control(RIGHT, 0.0, FALSE))
    return(ERROR);
  /* enable digital loopback mode with no loopback attenuation             */
  if (codec_loopback_enable(0.0))
    return(ERROR);
  return(OK);
}

/*************************************************************************/
/* CodecExBlkCapPb –  CODEC Example: Block Capture and Playback          */
/*                                                                       */
/*   This function provides an example of capturing a block of audio     */
/*   data and playing it back for verification                           *.
/*                                                                       */
/*************************************************************************/
int CodecExBlkCapPb(
         int        micSel,
         Mcbsp_dev  dev
                   )
{
  Mcbsp_config  mcbspConfig;
  unsigned char *pBuffer;
  int           sampleRate;
  unsigned int  numBytes;
  int           status;
  if (micSel)
    sampleRate= VOICE_BAND_SAMPLE_RATE;
  else
    sampleRate= PRO_AUDIO_SAMPLE_RATE;
  /*************************************************************************/
  /* allocate buffer for capture data                                      */
  /*************************************************************************/
  numBytes= BLOCK_CAPTURE_TIME * sampleRate * sizeof(int);
  /*-----------------------------------------------------------------------*/
  /* With USE_MALLOC set to 1, SDRAM will be used for the buffer           */
  /* With USE_MALLOC set to 0, SBSRAM will be used for the buffer          */
  /*   Note that SBSRAM will not hold much at the higher sample rate       */
  /*-----------------------------------------------------------------------*/
#undef  USE_MALLOC
#define USE_MALLOC  1
#if USE_MALLOC
  pBuffer = malloc( numBytes );
#else
  {
  /* Explicitly use SBSRAM                                                 */
    extern unsigned int SbsramDataSize, SbsramDataAddr;
    if ( numBytes <= (unsigned int)(&SbsramDataSize) )
```

```
    { pBuffer = (unsigned char *)(&SbsramDataAddr); }
    else
    { pBuffer = NULL; }
  }
#endif
  if (pBuffer == NULL)
  {
    DEBUG("Error allocating capture buffer in codecExBlkCapPb()");
    return(ERROR);
  }
  /*************************************************************************/
  /* configure McBSP                                                       */
  /*************************************************************************/
  memset(&mcbspConfig,0,sizeof(mcbspConfig));
  mcbspConfig.loopback              = FALSE;
  mcbspConfig.tx.update             = TRUE;
  mcbspConfig.tx.clock_polarity     = CLKX_POL_RISING;
  mcbspConfig.tx.frame_sync_polarity= FSYNC_POL_HIGH;
  mcbspConfig.tx.clock_mode         = CLK_MODE_EXT;
  mcbspConfig.tx.frame_sync_mode    = FSYNC_MODE_EXT;
  mcbspConfig.tx.phase_mode         = SINGLE_PHASE;
  mcbspConfig.tx.frame_length1      = 0;
  mcbspConfig.tx.word_length1       = WORD_LENGTH_32;
  mcbspConfig.tx.frame_ignore       = NO_FRAME_IGNORE;
  mcbspConfig.tx.data_delay         = DATA_DELAY0;
  mcbspConfig.rx.update             = TRUE;
  mcbspConfig.rx.clock_polarity     = CLKR_POL_FALLING;
  mcbspConfig.rx.frame_sync_polarity= FSYNC_POL_HIGH;
  mcbspConfig.rx.clock_mode         = CLK_MODE_EXT;
  mcbspConfig.tx.frame_sync_mode    = FSYNC_MODE_EXT;
  mcbspConfig.rx.phase_mode         = SINGLE_PHASE;
  mcbspConfig.rx.frame_length1      = 0;
  mcbspConfig.rx.word_length1       = WORD_LENGTH_32;
  mcbspConfig.rx.frame_ignore       = NO_FRAME_IGNORE;
  mcbspConfig.rx.data_delay         = DATA_DELAY0;
  EXIT_ERROR(mcbsp_config(dev,&mcbspConfig));
  /*************************************************************************/
  /* configure CODEC                                                       */
  /*************************************************************************/
  EXIT_ERROR(codec_init());
  codec_change_sample_rate(sampleRate, TRUE);
  if (micSel)
  {
    /* A/D 0.0 dB gain, turn on 20dB mic gain, sel (L/R)MIC as inputs      */
    EXIT_ERROR(codec_adc_control(LEFT,0.0,TRUE,MIC_SEL));
    EXIT_ERROR(codec_adc_control(RIGHT,0.0,TRUE,MIC_SEL));
#if PRINT_DBG
    printf("Begin speaking into the microphone when the LED illuminates.\n    ");
#endif
  }
  else
  {
    /* A/D 0.0 dB gain, turn off 20dB mic gain, sel (L/R)LINE input        */
```

```
    EXIT_ERROR(codec_adc_control(LEFT,0.0,FALSE,LINE_SEL));
    EXIT_ERROR(codec_adc_control(RIGHT,0.0,FALSE,LINE_SEL));
    /* mute (L/R)LINE input to mixer                                          */
    EXIT_ERROR(codec_line_in_control(LEFT,MIN_AUX_LINE_GAIN,TRUE));
    EXIT_ERROR(codec_line_in_control(RIGHT,MIN_AUX_LINE_GAIN,TRUE));
#if PRINT_DBG
    printf("Play input source when LED illuminates.\n     ");
#endif
  }
  /************************************************************************/
  /* begin capture process                                                */
  /************************************************************************/
  delay_msec(1000);         /* wait a second for user to view LED          */
  evm_led_enable(0);
  evm_led_enable(1);
  status= mcbsp_sync_receive(dev,pBuffer,numBytes,FALSE,NULL,FALSE);
  evm_led_disable(1);
  evm_led_disable(0);
  EXIT_ERROR(status);
  /************************************************************************/
  /* begin playback process                                               */
  /************************************************************************/
  delay_msec(1000);         /* pause                                       */
  /* D/A 0.0 dB atten, do not mute DAC outputs                              */
  EXIT_ERROR(codec_dac_control(LEFT, 0.0, FALSE));
  EXIT_ERROR(codec_dac_control(RIGHT, 0.0, FALSE));
  evm_led_enable(1);
  evm_led_enable(0);
  status = mcbsp_sync_send(dev,pBuffer,numBytes,FALSE,NULL,FALSE);
  evm_led_disable(1);
  evm_led_disable(0);
  EXIT_ERROR(status);
  /* mute DAC outputs                                                       */
  EXIT_ERROR(codec_dac_control(LEFT, 0.0, TRUE));
  EXIT_ERROR(codec_dac_control(RIGHT, 0.0, TRUE));
#if USE_MALLOC
  free(pBuffer);
#endif
  return(OK);
exit_err:
#if USE_MALLOC
  free(pBuffer);
#endif
  return(ERROR);

}
/************************************************************************/
/* CodecExContCapPb –  CODEC Example: Continuous Capture and Playback   */
/*                                                                      */
/*   This function provides an example of capturing into and playing back   */
/*   out of a buffer simultaneously.                                    */
/*                                                                      */
/************************************************************************/
```

```
int CodecExContCapPb(
         int        micSel,
         Mcbsp_dev  dev
                   )
{
  Mcbsp_config  mcbspConfig;
  unsigned char *pBuffer1;
  int           sampleRate;
  unsigned int  numBytes;
  int           status;
  if (micSel)
    sampleRate= VOICE_BAND_SAMPLE_RATE;
  else
    sampleRate= PRO_AUDIO_SAMPLE_RATE;
  /*************************************************************************/
  /* allocate buffer for capture data                                      */
  /*************************************************************************/
  /* see HEAP_SZ in makefile */
  numBytes= CONT_CAPTURE_TIME * sampleRate * sizeof(int);
  /*-----------------------------------------------------------------------*/
  /* With USE_MALLOC set to 1, SDRAM will be used for the buffer           */
  /* With USE_MALLOC set to 0, SBSRAM will be used for the buffer          */
  /*-----------------------------------------------------------------------*/
#undef  USE_MALLOC
#define USE_MALLOC  1
#if USE_MALLOC
  pBuffer1 = malloc( numBytes );
#else
  {
  /* Explicitly use SBSRAM                                                 */
    extern unsigned int SbsramDataSize, SbsramDataAddr;
    if ( numBytes <= (unsigned int)(&SbsramDataSize) )
    { pBuffer1 = (unsigned char *)(&SbsramDataAddr); }
    else
    { pBuffer1 = NULL; }
  }
#endif
  if (pBuffer1 == NULL)
  {
    DEBUG("Error allocating capture buffer 1 in codecExContCapPb()");
    return(ERROR);
  }
  /*************************************************************************/
  /* configure McBSP                                                       */
  /*************************************************************************/
  memset(&mcbspConfig,0,sizeof(mcbspConfig));
  mcbspConfig.loopback              = FALSE;
  mcbspConfig.tx.update             = TRUE;
  mcbspConfig.tx.clock_polarity     = CLKX_POL_RISING;
  mcbspConfig.tx.frame_sync_polarity= FSYNC_POL_HIGH;
  mcbspConfig.tx.clock_mode         = CLK_MODE_EXT;
  mcbspConfig.tx.frame_sync_mode    = FSYNC_MODE_EXT;
  mcbspConfig.tx.phase_mode         = SINGLE_PHASE;
```

```
  mcbspConfig.tx.frame_length1      = 0;
  mcbspConfig.tx.word_length1       = WORD_LENGTH_32;
  mcbspConfig.tx.frame_ignore       = FRAME_IGNORE;
  mcbspConfig.tx.data_delay         = DATA_DELAY0;
  mcbspConfig.rx.update             = TRUE;
  mcbspConfig.rx.clock_polarity     = CLKR_POL_FALLING;
  mcbspConfig.rx.frame_sync_polarity= FSYNC_POL_HIGH;
  mcbspConfig.rx.clock_mode         = CLK_MODE_EXT;
  mcbspConfig.tx.frame_sync_mode    = FSYNC_MODE_EXT;
  mcbspConfig.rx.phase_mode         = SINGLE_PHASE;
  mcbspConfig.rx.frame_length1      = 0;
  mcbspConfig.rx.word_length1       = WORD_LENGTH_32;
  mcbspConfig.rx.frame_ignore       = FRAME_IGNORE;
  mcbspConfig.rx.data_delay         = DATA_DELAY0;
  EXIT_ERROR(mcbsp_config(dev,&mcbspConfig));
  /*************************************************************************/
  /* configure CODEC                                                       */
  /*************************************************************************/
  EXIT_ERROR(codec_init());
  codec_change_sample_rate(sampleRate, TRUE);
  if (micSel)
  {
    /* A/D 0.0 dB gain, turn on 20dB mic gain, sel (L/R)MIC as inputs      */
    EXIT_ERROR(codec_adc_control(LEFT,0.0,TRUE,MIC_SEL));
    EXIT_ERROR(codec_adc_control(RIGHT,0.0,TRUE,MIC_SEL));
#if PRINT_DBG
    printf("Begin speaking into the microphone when the LED illuminates.\n    ");
#endif
  }
  else
  {
    /* A/D 0.0 dB gain, turn off 20dB mic gain, sel (L/R)LINE input         */
    EXIT_ERROR(codec_adc_control(LEFT,0.0,FALSE,LINE_SEL));
    EXIT_ERROR(codec_adc_control(RIGHT,0.0,FALSE,LINE_SEL));
    /* mute (L/R)LINE input to mixer                                       */
    EXIT_ERROR(codec_line_in_control(LEFT,MIN_AUX_LINE_GAIN,TRUE));
    EXIT_ERROR(codec_line_in_control(RIGHT,MIN_AUX_LINE_GAIN,TRUE));
#if PRINT_DBG
    printf("Play on input source when LED illuminates.\n    ");
#endif
  }

  /* D/A 0.0 dB atten, do not mute DAC outputs                             */
  EXIT_ERROR(codec_dac_control(LEFT, 0.0, FALSE));
  EXIT_ERROR(codec_dac_control(RIGHT, 0.0, FALSE));
  /* initialize globals */
  numBuffsPlayed = 0;
  delay_msec(1000);
  while (numBuffsPlayed < NUM_BUFFS_TO_PLAY)
  {
    if (numBuffsPlayed % 2)
    {
      evm_led_disable(1);
```

```
      evm_led_disable(0);
    }
    else
    {
      evm_led_enable(1);
      evm_led_enable(0);
    }
  /*************************************************************************/
  /* begin capture process                                                 */
  /*************************************************************************/
    status= mcbsp_sync_receive(dev, pBuffer1, numBytes, FALSE, NULL, FALSE);
    EXIT_ERROR(status);
  /*************************************************************************/
  /* begin playback process                                                */
  /*************************************************************************/
    status= mcbsp_async_send(dev,pBuffer1,numBytes,FALSE,NULL,AsyncTxCallback);
    EXIT_ERROR(status);
  } /* close while loop */
  /* mute DAC outputs                                                      */
  EXIT_ERROR(codec_dac_control(LEFT, 0.0, TRUE));
  EXIT_ERROR(codec_dac_control(RIGHT, 0.0, TRUE));
#if USE_MALLOC
  free(pBuffer1);
#endif
  evm_led_disable(1);
  evm_led_disable(0);
  return(OK);
exit_err:
#if USE_MALLOC
  free(pBuffer1);
#endif
  return(ERROR);

}
void AsyncTxCallback(Mcbsp_dev dev, int status)
{
  numBuffsPlayed++;
  return;
}
/*************************************************************************/
/* CodecExContTonePb() – Play continuous tone                            */
/*   This function provides an example of playing back a buffer of data in   */
/*   continuous (dma autoinitialization) mode.                           */
/*                                                                       */
/*************************************************************************/
int CodecExContTonePb( Mcbsp_dev dev, int leftRightSel, int duration )
{
  Mcbsp_config   mcbspConfig;
  int           numBytes;
  int           sampleRate;
  int           status;
  int           i;
  short         *pBuffer;
```

```
  sampleRate = TONE_PB_SAMPLE_RATE;
  /*************************************************************************/
  /* allocate buffer for audio data                                        */
  /*************************************************************************/
  numBytes = TONE_PB_SAMPLE_RATE * sizeof(int);
  /*-----------------------------------------------------------------------*/
  /* With USE_MALLOC set to 1, SDRAM will be used for the buffer           */
  /* With USE_MALLOC set to 0, SBSRAM will be used for the buffer          */
  /*-----------------------------------------------------------------------*/
#undef  USE_MALLOC
#define USE_MALLOC  0
#if USE_MALLOC
  pBuffer = (short *)(malloc( numBytes ));
#else
  {
  /* Explicitly use SBSRAM                                                 */
    extern unsigned int SbsramDataSize, SbsramDataAddr;
    if ( numBytes <= (unsigned int)(&SbsramDataSize) )
    { pBuffer = (short *)(&SbsramDataAddr); }
    else
    { pBuffer = NULL; }
  }
#endif
  if (pBuffer == NULL)
  {
    DEBUG("Error allocating capture buffer in codecExContTonePb()");
    return(ERROR);
  }

  /*************************************************************************/
  /* synthesize tone                                                       */
  /*************************************************************************/
  GenTone(pBuffer,numBytes/2,(float)TONE_FREQ,(float)sampleRate,leftRightSel);
  /*************************************************************************/
  /* configure mcbsp transmitter                                           */
  /*************************************************************************/

  memset(&mcbspConfig,0,sizeof(mcbspConfig));
  mcbspConfig.loopback              = FALSE;
  mcbspConfig.tx.update             = TRUE;
  mcbspConfig.tx.clock_polarity     = CLKX_POL_RISING;
  mcbspConfig.tx.frame_sync_polarity= FSYNC_POL_HIGH;
  mcbspConfig.tx.clock_mode         = CLK_MODE_EXT;
  mcbspConfig.tx.frame_sync_mode    = FSYNC_MODE_EXT;
  mcbspConfig.tx.phase_mode         = SINGLE_PHASE;
  mcbspConfig.tx.frame_length1      = 0;
  mcbspConfig.tx.word_length1       = WORD_LENGTH_32;
  mcbspConfig.tx.frame_ignore       = NO_FRAME_IGNORE;
  mcbspConfig.tx.data_delay         = DATA_DELAY0;
  mcbspConfig.rx.update             = FALSE;
  mcbspConfig.srg.update            = FALSE;
  EXIT_ERROR(mcbsp_config(dev,&mcbspConfig));
```

```
  /*************************************************************************/
  /* configure CODEC                                                       */
  /*************************************************************************/
  EXIT_ERROR(codec_init());
  codec_change_sample_rate(sampleRate, TRUE);
  /* D/A 0.0 dB atten, do not mute DAC outputs                             */
  EXIT_ERROR(codec_dac_control(LEFT, 0.0, FALSE));
  EXIT_ERROR(codec_dac_control(RIGHT, 0.0, FALSE));
  numBuffsPlayed= 0;
  evm_led_enable(1);
  evm_led_enable(0);
  status = mcbsp_cont_async_send(dev,(unsigned char *)pBuffer,NULL,
                                 numBytes,FALSE,NULL,ContTonePbCallback );
  while(1)
  {
      delay_msec(10);
      if ( numBuffsPlayed < (duration – 1) )
      { continue; }
      else
      { dev->tx_dma.continuous= FALSE; }
      if ( numBuffsPlayed >= duration )
      { break; }
  }
  evm_led_disable(1);
  evm_led_disable(0);
  /* mute DAC outputs                                                      */
  EXIT_ERROR(codec_dac_control(LEFT, 0.0, TRUE));
  EXIT_ERROR(codec_dac_control(RIGHT, 0.0, TRUE));

  delay_msec(1000);
#if USE_MALLOC
  free(pBuffer);
#endif
  return(OK);
exit_err:
#if USE_MALLOC
  free(pBuffer);
#endif
  return(ERROR);
}
void ContTonePbCallback(Mcbsp_dev dev,int status)
{
  numBuffsPlayed++;
  return;
}
/*************************************************************************/
/* GenTone() – Generate tone samples with indicated frequency.
/*
/*************************************************************************/
int GenTone(                          /* RET: OK or ERROR                */
            short         *pBuffer,   /* I/O: buffer for generated data  */
            int           buffSize,   /*  IN: number of shorts in buffer */
            float         toneFreq,   /*  IN: requested tone frequency (Hz) */
```

```
            float         sampleRate,  /*  IN: current CODEC sample rate (Hz) */
            int           leftRightSel)/*  IN: LEFT, RIGHT, or BOTH           */
{
  int i;
  float omega;
  float deltaT;
  short scaledVal;
  memset((unsigned char *)pBuffer,0,(buffSize * sizeof(short)));
  for (i=0; i< buffSize; i++)
  {
    omega= 2 * PI * toneFreq;
    deltaT= (i/2) / sampleRate;
    scaledVal= (short)(32767 * (sinf(omega*deltaT)));

    if ( (leftRightSel == RIGHT) || (leftRightSel == BOTH) )
      pBuffer[i] = scaledVal;
    i++;
    if ( (leftRightSel == LEFT) || (leftRightSel == BOTH) )
      pBuffer[i]   = scaledVal;
  }
  return(0);
}
```

(b) codec_ex.h file

```
/*****************************************************************************/
/*   CODEC_EX.H – Codec examples header file                                 */
/*                                                                           */
/*     This is the public header file for the codec examples for the EVM     */
/*     board.                                                                */
/*                                                                           */
/*  FUNCTIONS:                                                               */
/*    CodecExLb()         –  CODEC Example: Loopback mode                    */
/*    CodecExBlkCapPb()   –  CODEC Example: Block Capture and Playback       */
/*    CodecExContTonePb() –  CODEC Example: Continuous Tone Playback         */
/*    CodecExContCapPb()  –  CODEC Example: Continuous Capture and Playback  */
/*                                                                           */
/*  MACRO FUNCTIONS:                                                         */
/*                                                                           */
/*****************************************************************************/
#ifndef _CODEC_EX_H_
#define _CODEC_EX_H_
/*---------------------------------------------------------------------------*/
/* INCLUDES                                                                  */
/*---------------------------------------------------------------------------*/
/*---------------------------------------------------------------------------*/
/* DEFINES AND MACROS                                                        */
/*---------------------------------------------------------------------------*/
/*---------------------------------------------------------------------------*/
/* GLOBAL VARIABLES                                                          */
/*---------------------------------------------------------------------------*/
/*---------------------------------------------------------------------------*/
/* FUNCTIONS                                                                 */
/*---------------------------------------------------------------------------*/
/*****************************************************************************/
/* CodecExLb –  CODEC Example: Loopback mode                                 */
/*                                                                           */
/*   This function provides an example of using the CS4321A CODEC in         */
/*   Digital Loopback Mode (DLB).                                            */
/*                                                                           */
/*****************************************************************************/
int CodecExLb(                  /* RET: OK or ERROR                          */
         int micSel             /* IN : TRUE – mic sel, FALSE – line sel     */
             );
/*****************************************************************************/
/* CodecExBlkCapPb –  CODEC Example: Block Capture and Playback              */
/*                                                                           */
/*   This function provides an example of capturing a block of audio         */
/*   data and playing it back for verification                                */
/*                                                                           */
/*****************************************************************************/
int CodecExBlkCapPb(
         int        micSel,
         Mcbsp_dev  dev
                   );
```

```
/*************************************************************************/
/* CodecExContTonePb() – Play continuous tone                            */
/*   This function provides an example of playing back a buffer of data in   */
/*   continuous (dma autoinitialization) mode.                           */
/*                                                                       */
/*************************************************************************/
int CodecExContTonePb( Mcbsp_dev dev, int leftRightSel, int duration );
/*************************************************************************/
/* CodecExContCapPb() – Capture and playback simultaneously              */
/*************************************************************************/
int CodecExContCapPb(
         int        micSel,
         Mcbsp_dev  dev
                  );
#endif
```

(c) main.c file

```
#include <stdio.h>
#include <string.h>
#include <common.h>
#include <mcbspdrv.h>
#include <board.h>
#include <codec.h>
#include ”codec_ex.h”
#define PRINT_DBG 0
static void led_blink( int count, int ms_period );
main()
{
  int status;
  Mcbsp_dev dev;
  int i;
  /*************************************************************************/
  /* Initialize EVM                                                        */
  /*************************************************************************/
  if (evm_init())
  {
    printf(”ERROR returned from evm_init()\n”);
    return(ERROR);
  }

  led_blink( 2, 1000 );
  /*************************************************************************/
  /* Codec Loopback Example                                                */
  /*************************************************************************/
#if 1
#if PRINT_DBG
  printf(”\nCodec Line In Loopback example.   \n”);
#endif
  status= CodecExLb( FALSE );
  if (status == ERROR)
  {
    printf(”Error calling CodecExLb()\n”);
    return(ERROR);
  }
  delay_msec(10000);
  led_blink( 10, 200 );
#endif
  /*************************************************************************/
  /* Open MCBSP for subsequent Examples                                    */
  /*************************************************************************/
  mcbsp_drv_init();
  dev= mcbsp_open(0);
  if (dev == NULL)
  {
    printf(”Error opening MCBSP 0    \n  ”);
    return(ERROR);
  }
#if 1
  /*************************************************************************/
  /* Single Block Capture and Playback Example                             */
  /*************************************************************************/
```

```
#if PRINT_DBG
  printf("Codec Block Capture Playback example.   \n");
#endif
  status = CodecExBlkCapPb( FALSE, dev );
  if (status == ERROR)
  {
    printf("Error calling CodecExBlkCapPb()\n  ");
    return(ERROR);
  }
  led_blink( 10, 200 );
#endif
#if 1
  /*****************************************************************************/
  /* Codec simultaneous capture and playback over McBSP Example                */
  /*****************************************************************************/
#if PRINT_DBG
  printf("Codec Continuous Capture Playback example.   \n");
#endif
  status = CodecExContCapPb( TRUE, dev );
  if (status == ERROR)
  {
    printf("Error calling CodecExContCapPb() \n  " );
    return(ERROR);
  }
  led_blink( 10, 200 );
#endif
  /*****************************************************************************/
  /* Codec Tone Generator using DMA autoinitialization Example                 */
  /*****************************************************************************/
#if 1
#if PRINT_DBG
  printf("Codec Continuous Tone Playback example.   \n");
#endif
  status = CodecExContTonePb( dev, BOTH, 5 );
  if (status == ERROR)
  {
    printf("Error calling CodecExContTonePb()\n  ");
    return(ERROR);
  }
  led_blink( 10, 200 );
#endif
  mcbsp_close(dev);
  printf("Thats All Folks\n");
  led_blink( 100, 100 );
  return(OK);
}
static void led_blink( int count, int ms_period )
{
    int     i;
    for (i=0;i<count;i++)
    {
        evm_led_enable(0); evm_led_enable(1); delay_msec(ms_period/2);
        evm_led_disable(0); evm_led_disable(1); delay_msec(ms_period/2);
    }
}
```

(d) link.cmd file

```
/* linker command file for examples (MAP 1) */
MEMORY
{
  INT_PROG_MEM (RX)    : origin = 0x00000000 length = 0x00010000
  SBSRAM_PROG_MEM (RX) : origin = 0x00400000 length = 0x00014000
  SBSRAM_DATA_MEM (RW) : origin = 0x00414000 length = 0x0002C000
  SDRAM0_DATA_MEM (RW) : origin = 0x02000000 length = 0x00400000
  SDRAM1_DATA_MEM (RW) : origin = 0x03000000 length = 0x00400000
  INT_DATA_MEM (RW)    : origin = 0x80000000 length = 0x00010000
}
SECTIONS
{
  .vec:          load = 0x00000000
  .text:         load = SBSRAM_PROG_MEM
  .const:        load = INT_DATA_MEM
  .bss:          load = INT_DATA_MEM
  .data:         load = INT_DATA_MEM
  .cinit         load = INT_DATA_MEM
  .pinit         load = INT_DATA_MEM
  .stack         load = INT_DATA_MEM
  .far           load = INT_DATA_MEM
  .sysmem        load = SDRAM0_DATA_MEM
  .cio           load = INT_DATA_MEM
  sbsbuf         load = SBSRAM_DATA_MEM
                 { _SbsramDataAddr = .; _SbsramDataSize = 0x0002C000; }
}
```

Appendix A

TMS320C6x EVM Connector Pinouts

This appendix contains the pinout information for each connector on the TMS320C6x EVM.

Topic **Page**

A.1 TMS320C6x EVM Connector Summary

There are ten connectors on the ’C6x EVM, as shown in Table A–1. The J4 CPLD ISP connector is for factory use and is not installed.

Table A–1. TMS320C6x EVM Connectors Summary

Connector	No. of Pins	Description	Type	See Page
J1	3	Stereo microphone input	3.5-mm audio jack	A-2
J2	3	Stereo line input	3.5-mm audio jack	A-3
J3	3	Stereo line output	3.5-mm audio jack	A-3
J4	10	CPLD ISP header	2 × 5 pin, 0.1 in.	A-4
J5	14	’C6x JTAG emulation header	2 × 7 pin, 0.1 in.	A-5
J6	80	Expansion memory interface	2 × 40 pos, 0.050-in. SMT	A-6
J7	80	Expansion peripheral interface	2 × 40 pos., 0.050-in. SMT	A-7
J8	4	External power	Molex disk drive, right-angle	A-8
J9	2	DSP fan power	Molex 1.25-mm, right-angle	A-8
P1	124	PCI local bus	Edge connector	A-9

A.2 Stereo Microphone Input Jack

Connector J1 supports a stereo microphone input. If a mono microphone is used, then the input left (tip) channel is only used. Both microphone input channels provide a voltage bias for electret microphones. The microphone input is the top 3.5-mm jack on the EVM’s mounting bracket (adjacent to the status LED).

Table A–2. Stereo Microphone Input Connector J1 Pinout

J1 Pin	Signal Name	Description	Type
TIP	MIC_L	Microphone left input channel	I
RING	MIC_R	Microphone right input channel	I
SLEEVE	AGND	Audio ground	–

A.3 Stereo Line Input Jack

Connector J2 supports a stereo line input. The line input audio connector is the middle 3.5-mm jack on the EVM's mounting bracket.

Table A–3. Stereo Line Input Connector J2 Pinout

J2 Pin	Signal Name	Description	Type
TIP	LI_L	Line input left channel	I
RING	LI_R	Line input right channel	I
SLEEVE	AGND	Audio ground	–

A.4 Stereo Line Output Jack

Connector J3 supports a stereo line input. The line output audio connector is the bottom 3.5-mm jack on the EVM's mounting bracket.

Table A–4. Stereo Line Output Connector J3 Pinout

J3 Pin	Signal Name	Description	Type
TIP	LO_L	Line output left channel	O
RING	LO_R	Line output right channel	O
SLEEVE	AGND	Audio ground	–

A.5 CPLD ISP Header

Connector J4 provides the CPLD's JTAG in-system programming port that allows the EVM's onboard logic to be reprogrammed. This connector is a 10-pin header (two rows of five pins) with connections shown in Table A–5 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 EVM. The pins have 0.025-inch square posts with 0.100-inch spacing.

The J4 CPLD ISP connector is for factory use and is not installed.

Table A–5. CPLD ISP J4 Pinout

J4 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.6 TMS320C6x JTAG Emulation Header

Connector J5 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–6 to communicate with an XDS510 emulator. Pin 6 is used for keying to ensure a proper connection.

Table A–6. TMS320C6x JTAG Emulation Header J5 Pinout

J5 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 EVM.	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.7 Expansion Memory Interface Connector

Connector J6 provides the 'C6201 asynchronous expansion memory interface signals to a daughterboard that can provide additional memory and memory-mapped devices.

Table A–7. Expansion Memory Interface J6 Connector Pinout

J6 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	XBE3	O	28	XBE2	O
29	XBE1	O	30	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	XRE	O	74	XWE	O
75	XOE	O	76	XRDY	I
77	SPARE (N/C)	–	78	XCE1	O
79	GND	–	80	GND	–

A.8 Expansion Peripheral Interface Connector

Connector J7 provides ’C6201 expansion peripheral interface signals to a daughterboard.

Table A–8. Expansion Peripheral Interface J7 Connector Pinout

J7 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.9 External Power Connector

Connector J8 enables the ’C6x EVM to be connected to an external power supply during stand-alone operation.

Table A–9. External Power J8 Connector Pinout

J8 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.10 DSP Fan Power Connector

Connector J9 provides power to the DSP cooling fan. This 2-pin connector provides 5 V at 100 mA to the fan.

Table A–10. DSP Fan Power J9 Connector Pinout

J9 Pin No.	Signal Name	Description	Type
1	GND	Ground	–
2	PWR	5 V_{DC} at 100 mA	O

A.11 PCI Local Bus Connector

Connector P1 provides the PCI local bus to the ’C6x EVM.

Table A–11. 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 EVM Schematics

This appendix contains the schematics for the TMS320C6x EVM.

NOTES, UNLESS OTHERWISE SPECIFIED:

1. RESISTANCE VALUES ARE IN OHMS.

2. CAPACITANCE VALUES ARE IN MICROFARADS.

3. HIGHEST REFERENCE DESIGNATOR USED:
   BOTTOM SIDE COMPONENTS START AT REF DES 500

	TOP (B SIDE)	BOT (A SIDE)
A. CERAMIC CAPS	C53	C629
B. TANTALUM CAPS	CT23	
C. ELECTROLYTIC CAPS	CE4	
D. DIODES	D7	D503
E. CONNECTORS/HEADERS	J9	
F. FILTER	E1	
G. RESISTORS	R90 / RN6	R645
H. SWITCHES	SW2	
I. IC'S	U32	U504
J. INDUCTORS	L10	
K. CRYSTALS/OSCILLATORS	Y4	

4. PARTS NOT INSTALLED ARE INDICATED WITH 'NU'
   THE FOLLOWING IS THE LIST OF UNINSTALLED PARTS:
   D501, R55, R501, R503, R580, R582,
   R584, R587, R600, R607, R613,
   R616, R618, R619, R621, R628

5. R628 TO BE USED WITH 1.8V CORE VOLTAGE ONLY.
   R624 TO BE USED WITH 2.5V CORE VOLTAGE ONLY.

6. INSTALL R582 TO PERMANENTLY ENABLE U504.
   DO NOT INSTALL R581 WHEN INSTALLING R582.

7. R7 & R8 MAY BE REMOVED WHEN USING 1Vrms LINE INPUT LEVEL.

8. INSTALL R597 WITH 2.5V CORE VOLTAGE AND DO NOT INSTALL D501.
   WHEN USING 1.8V CORE VOLTAGE INSTALL D501 AND DO NOT INSTALL R597.

9. U22 MAY BE SUBSTITUTED WITH A PT6305B.

10. R584 & R580 ARE OPTIONAL TO ADJUST VCC3 OUTPUT VOLTAGE UP OR DOWN.
    RESISTORS NOT REQUIRED FOR NORMAL OPERATION.

REVISIONS

REV	DESCRIPTION	DATE	APPROVED

REVISION STATUS OF SHEETS

REV	*	*	*	*	*		
SH	36	37	38	39	40		
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

NEXT ASSY	D600830
USED ON	
APPLICATION	

1.	COVER SHEET	
2.	CLOCK-C6201 PLL & RESET	
3.	CLOCK-OSCILLATORS	
4.	CLOCK-FANOUT	
5.	EMIF-C6201	
6.	EMIF-BUS SWITCH	
7.	EMIF-ADDR BUFFERS BITS 12..2	
8.	EMIF-ADDR BUFFERS BITS 21..13	
9.	EMIF-PB TO ADD-ON BITS 7..0	
10.	EMIF-DATA XCVRS BITS 15..0	
11.	EMIF-DATA XCVRS BITS 31..16	
12.	EMIF-CONTROL BUFFERS	
13.	EMIF-SBSRAM	
14.	EMIF-SDRAM BANK 0	
15.	EMIF-SDRAM BANK 1	
16.	MCBSP-C6201 & MUX	
17.	AUDIO-CODEC	
18.	AUDIO-LINE IN & MIC	
19.	AUDIO-LINE OUT	
20.	HPI-C6201 & BUFFERS	
21.	JTAG-CONTROLLER	
22.	JTAG-HEADER & MUX	
23.	MISC-BOOTMODE & USER OPTIONS	
24.	PLD	
25.	PLD	
26.	PLD	
27.	PCI-CONNECTOR	
28.	PCI-CONTROLLER BUS INTERFACE	
29.	PCI-CONTROLLER ADD ON INTERFACE	
30.	DB-EXPANSION MEMORY INTERFACE	
31.	DB-EXPANSION PERIPHERAL INTERFACE	
32.	PWR-REGULATORS & RESET	
33.	PWR-ANALOG 5V & CONNECTORS	
34.	PWR-C6201	
35.	PWR-CAPS	
36.	PWR-CAPS	
37.	C6201 NC & THERMAL BALLS	
38.	TEST POINTS	
39.	TEST POINTS	
40.	TEST POINTS	

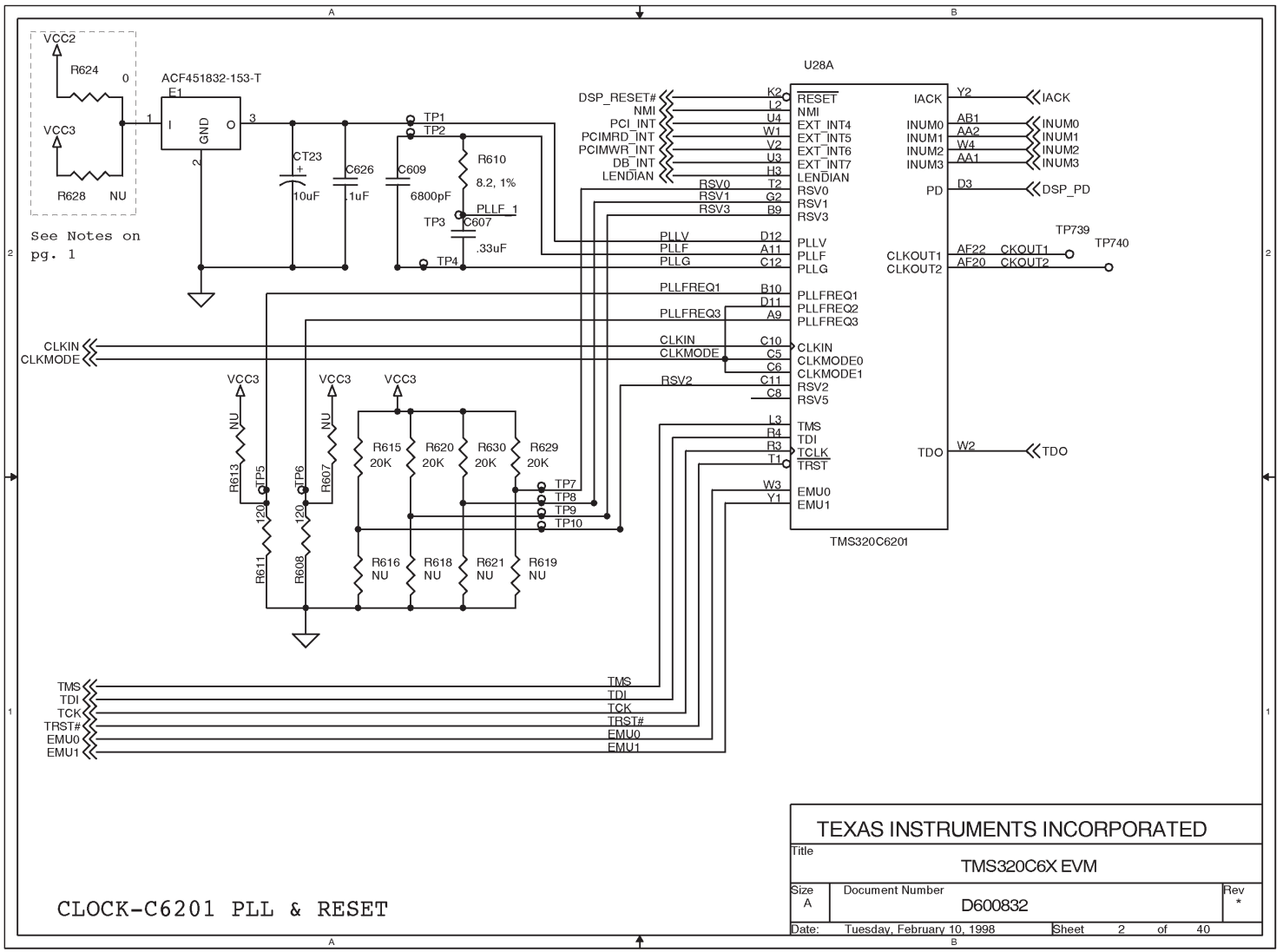
DWN	B. Dempsey	DATE
CHK	M. Dawkins	DATE
ENGR	G. Connelly	DATE
ENGR-MGR	T. Miscio	DATE
QA	J. Whisonant	DATE
MFG	M. Jackson	DATE
RLSE	J. Clark	DATE

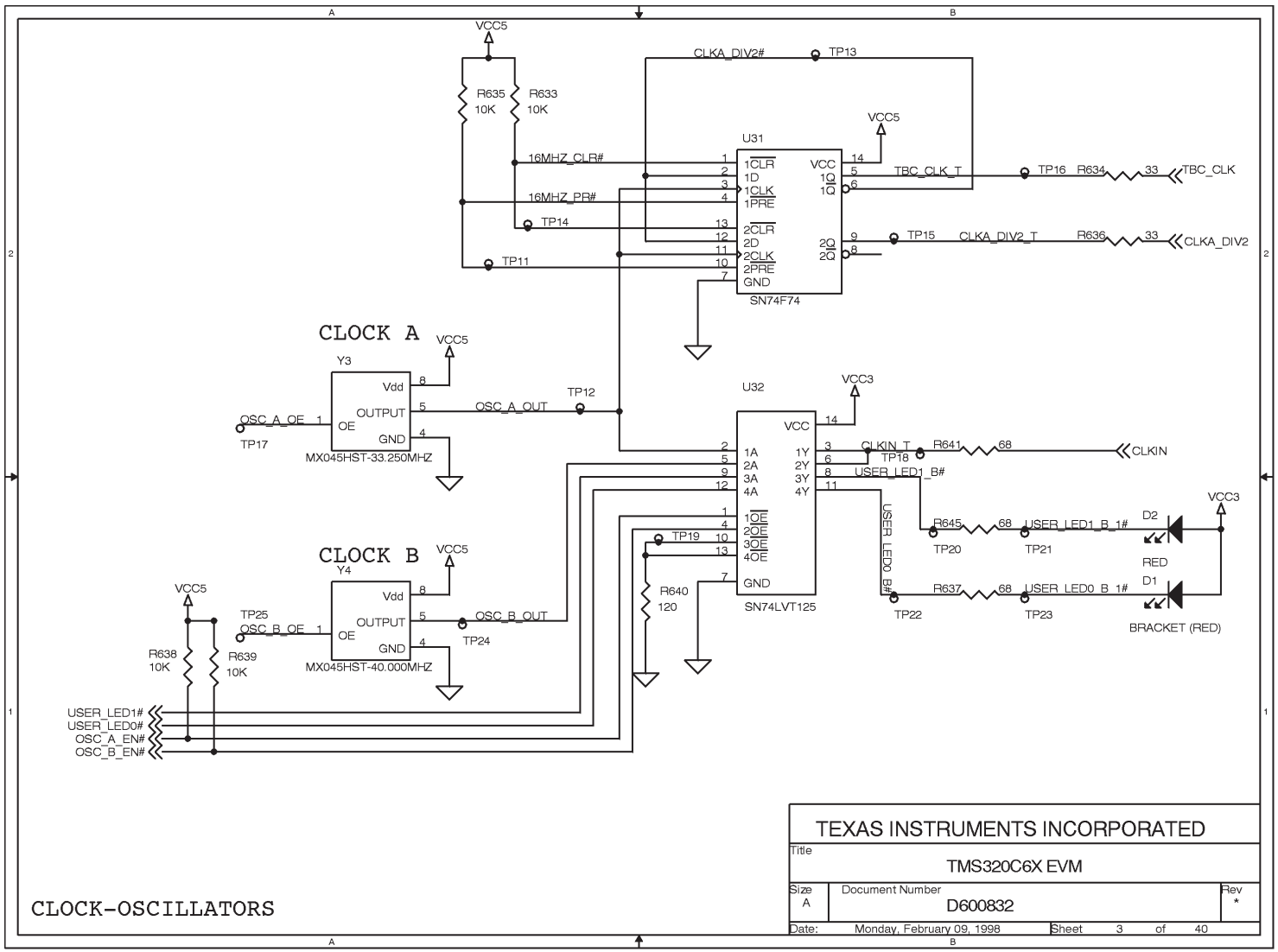
TEXAS INSTRUMENTS INCORPORATED
Software Development Systems, Semiconductor Group, Houston, Texas

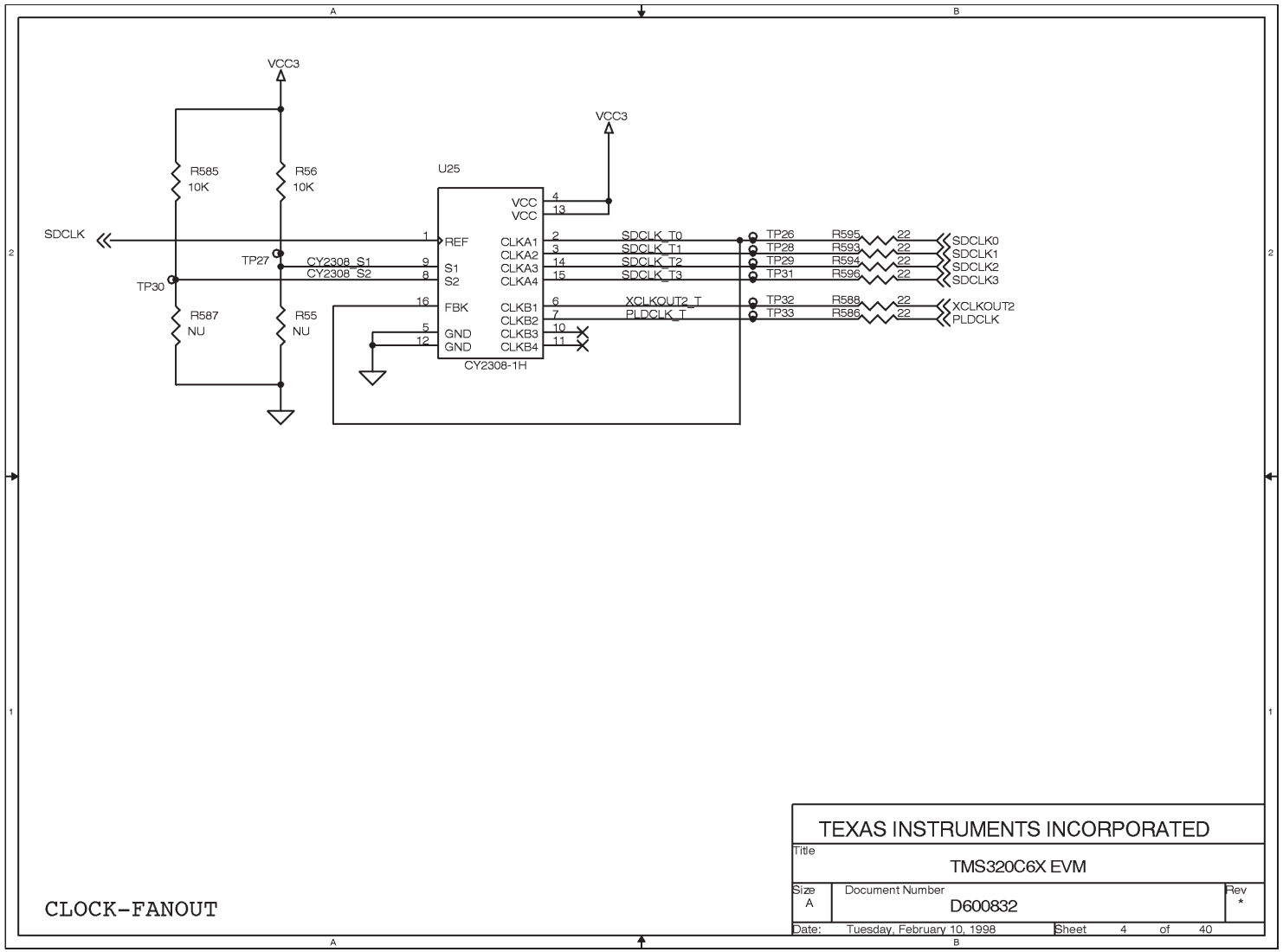
Title: TMS320C6X EVM

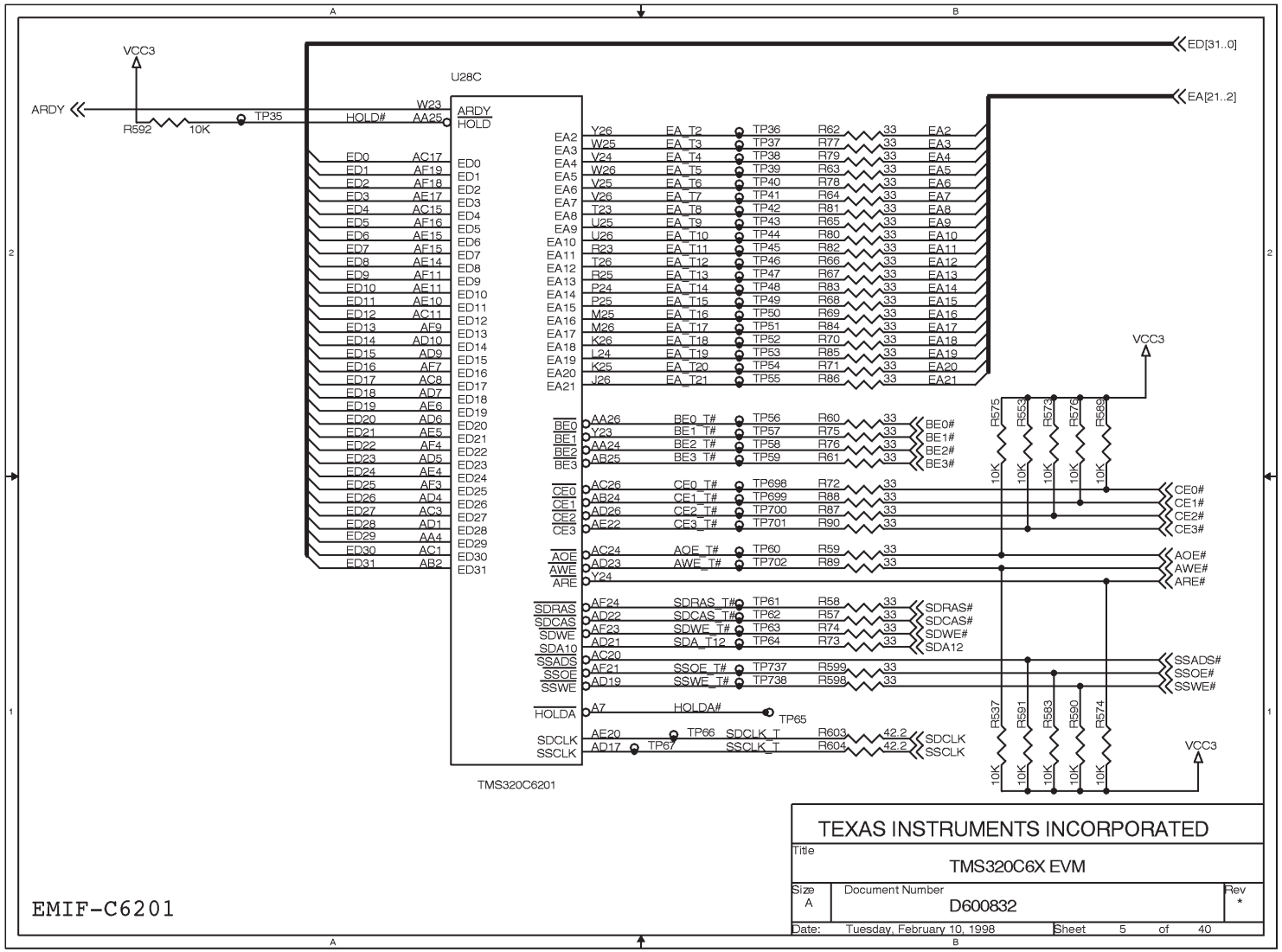
Size: A | Document Number: D600832 | Rev: *

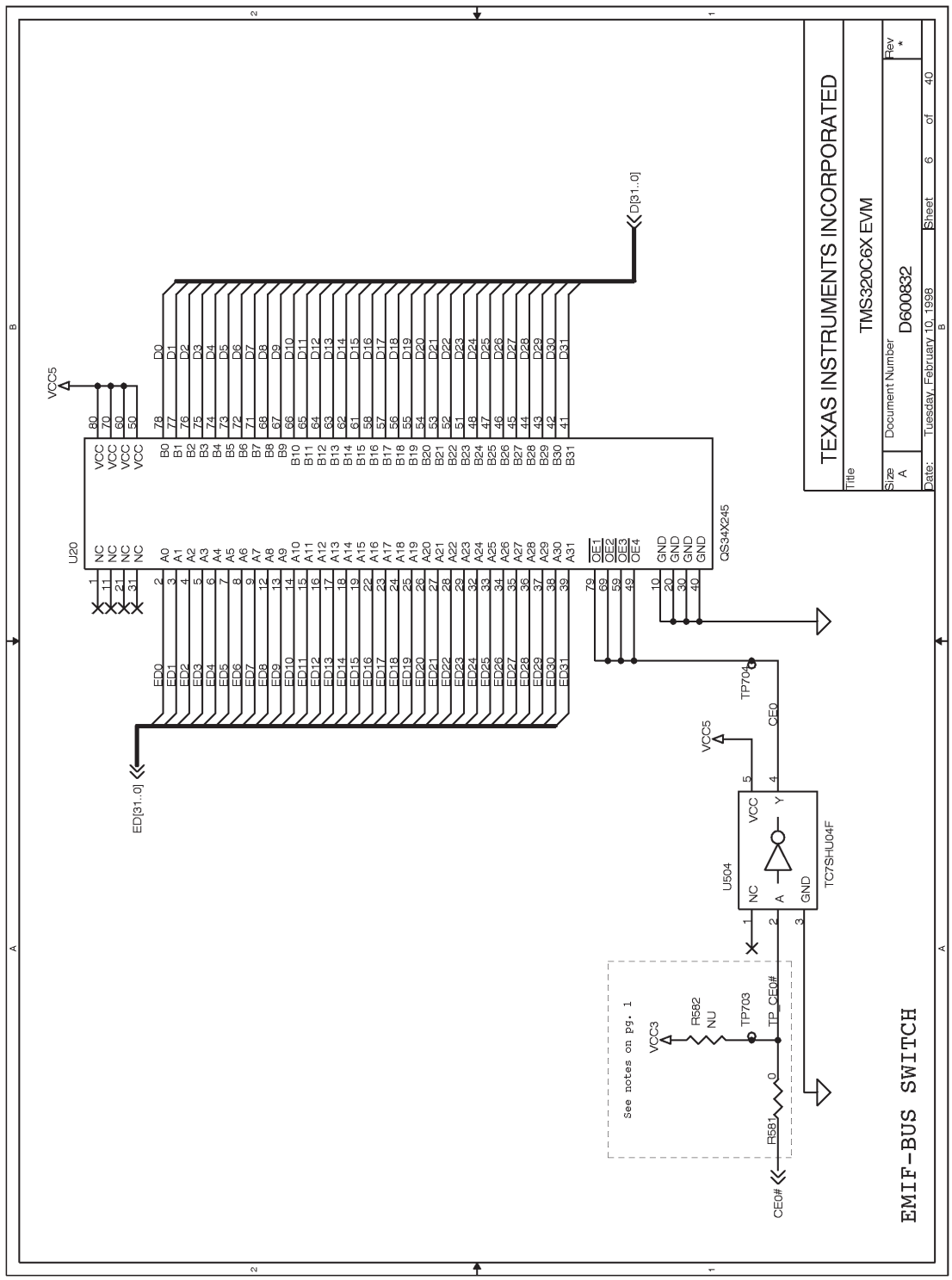
Date: Monday, February 09, 1998 | Sheet 1 of 40

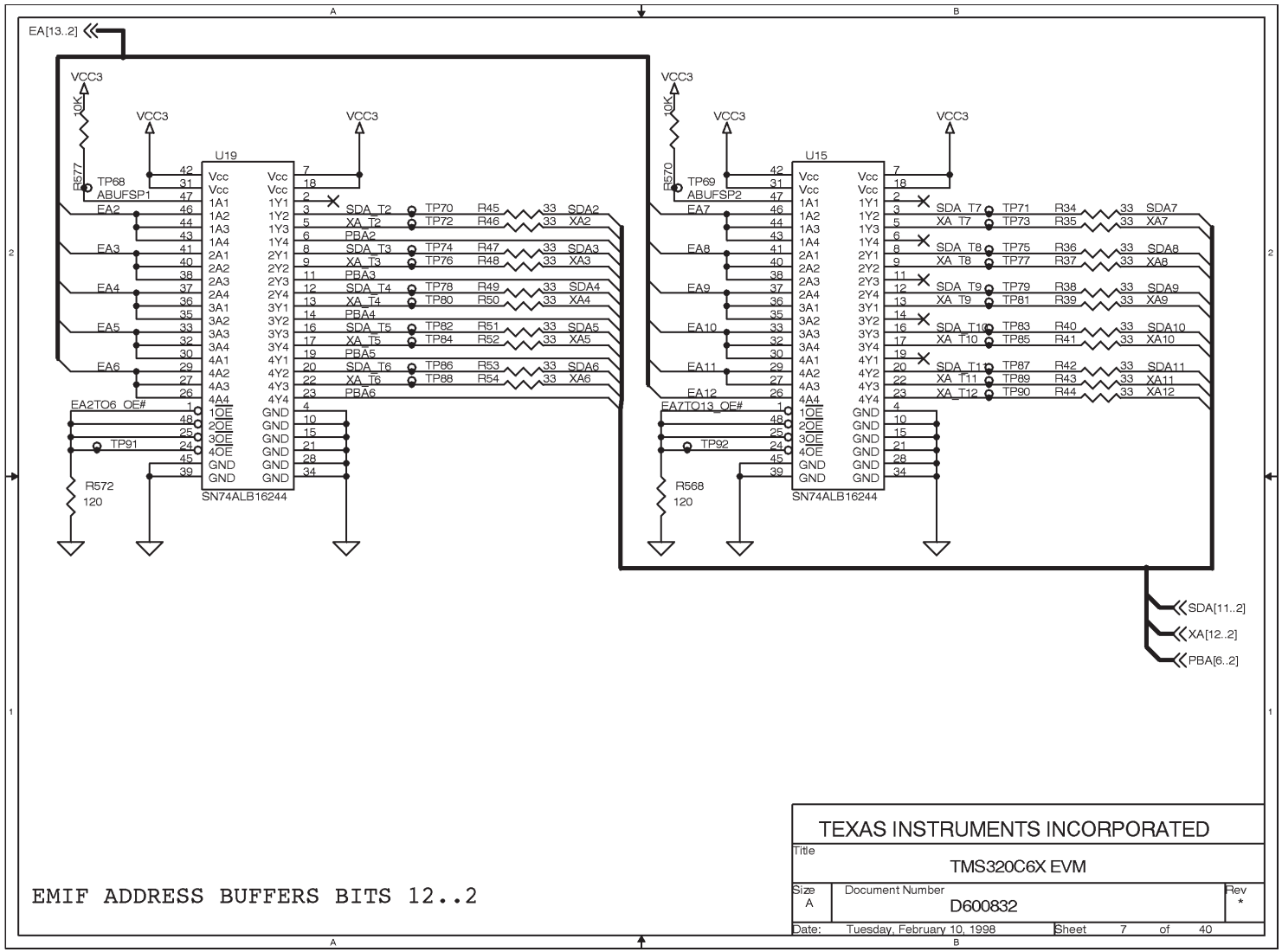

VCC2
R624
0
VCC3
R628
NU
ACF451832-153-T
E1
I
GND
O
See Notes on pg. 1
CT23
10uF
C626
.1uF
C609
6800pF
R610
8.2, 1%
PLLF_1
TP3
C607
.33uF
TP4
TP1
TP2
U28A
DSP_RESET#
NMI
PCI_INT
PCIMRD_INT
PCIMWR_INT
DB_INT
LENDIAN
RESET
EXT_INT4
EXT_INT5
EXT_INT6
EXT_INT7
LENDIAN
RSV0
RSV1
RSV3
PLLV
PLLF
PLLG
PLLFREQ1
PLLFREQ2
PLLFREQ3
CLKIN
CLKMODE0
CLKMODE1
RSV2
RSV5
TMS
TDI
TCLK
TRST
EMU0
EMU1
IACK
INUM0
INUM1
INUM2
INUM3
PD
CLKOUT1
CLKOUT2
TDO
TMS320C6201
TP739
TP740
CLKIN
CLKMODE
VCC3
R613
R607
R615
R620
R630
R629
20K
120
R611
R608
R616
R618
R621
R619
NU
TP5
TP6
TP7
TP8
TP9
TP10
TMS
TDI
TCK
TRST#
EMU0
EMU1
TEXAS INSTRUMENTS INCORPORATED
Title
TMS320C6X EVM
Size
A
Document Number
D600832
Rev
*
Date: Tuesday, February 10, 1998
Sheet 2 of 40
CLOCK-C6201 PLL & RESET


VCC5
R635
10K
R633
10K
16MHZ_CLR#
16MHZ_PR#
TP14
TP11
CLKA_DIV2#
TP13
U31
1CLR
1D
1CLK
1PRE
2CLR
2D
2CLK
2PRE
GND
VCC
1Q
2Q
SN74F74
TBC_CLK_T
TP16
R634
33
TBC_CLK
TP15
CLKA_DIV2_T
R636
33
CLKA_DIV2
CLOCK A
Y3
Vdd
OUTPUT
OE
GND
OSC_A_OE
TP17
OSC_A_OUT
TP12
MX045HST-33.250MHZ
U32
SN74LVT125
1A
2A
3A
4A
1Y
2Y
3Y
4Y
1OE
2OE
3OE
4OE
CLKIN_T
TP18
R641
68
CLKIN
USER_LED1_B#
USER_LED0_B#
R645
68
USER_LED1_B_1#
TP20
TP21
D2
RED
R637
68
USER_LED0_B_1#
TP22
TP23
D1
BRACKET (RED)
TP19
R640
120
CLOCK B
Y4
TP25
OSC_B_OE
OSC_B_OUT
TP24
MX045HST-40.000MHZ
R638
10K
R639
10K
USER_LED1#
USER_LED0#
OSC_A_EN#
OSC_B_EN#
TEXAS INSTRUMENTS INCORPORATED
Title
TMS320C6X EVM
Size
A
Document Number
D600832
Rev
*
Date:
Monday, February 09, 1998
Sheet
3
of
40
CLOCK-OSCILLATORS


VCC3
R585
10K
R56
10K
U25
VCC3
VCC
VCC
SDCLK
TP27
TP30
CY2308_S1
CY2308_S2
REF
S1
S2
FBK
GND
GND
CLKA1
CLKA2
CLKA3
CLKA4
CLKB1
CLKB2
CLKB3
CLKB4
CY2308-1H
SDCLK_T0
SDCLK_T1
SDCLK_T2
SDCLK_T3
XCLKOUT2_T
PLDCLK_T
TP26
TP28
TP29
TP31
TP32
TP33
R595
R593
R594
R596
R588
R586
22
SDCLK0
SDCLK1
SDCLK2
SDCLK3
XCLKOUT2
PLDCLK
R587
NU
R55
NU
CLOCK-FANOUT
TEXAS INSTRUMENTS INCORPORATED
Title
TMS320C6X EVM
Size
A
Document Number
D600832
Rev
*
Date: Tuesday, February 10, 1998
Sheet 4 of 40

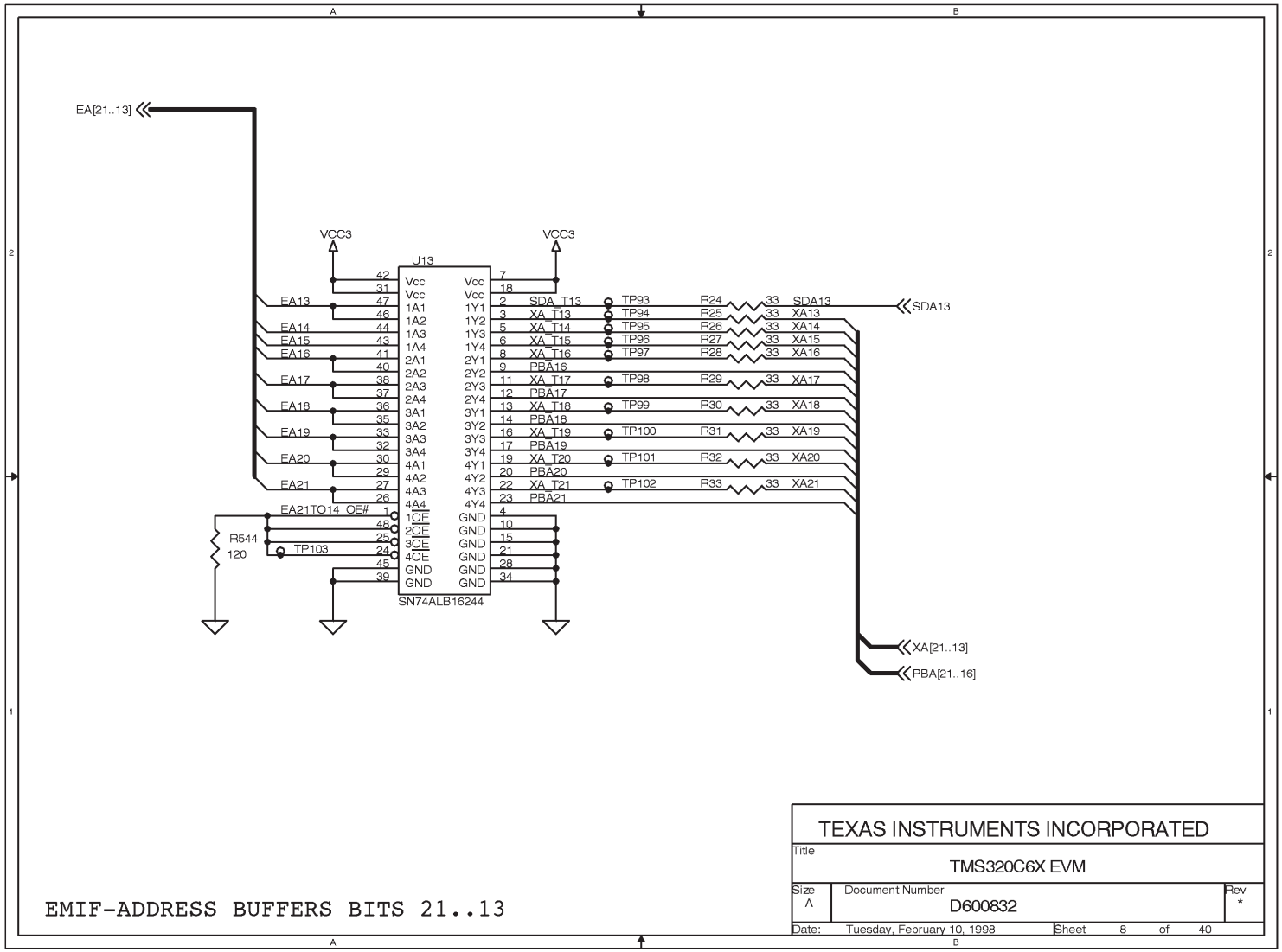

VCC3
ARDY
R592
10K
TP35
HOLD#
U28C
ARDY
HOLD
ED[31..0]
EA[21..2]
TMS320C6201
EMIF-C6201
TEXAS INSTRUMENTS INCORPORATED
Title
TMS320C6X EVM
Size
A
Document Number
D600832
Rev
*
Date: Tuesday, February 10, 1998
Sheet 5 of 40

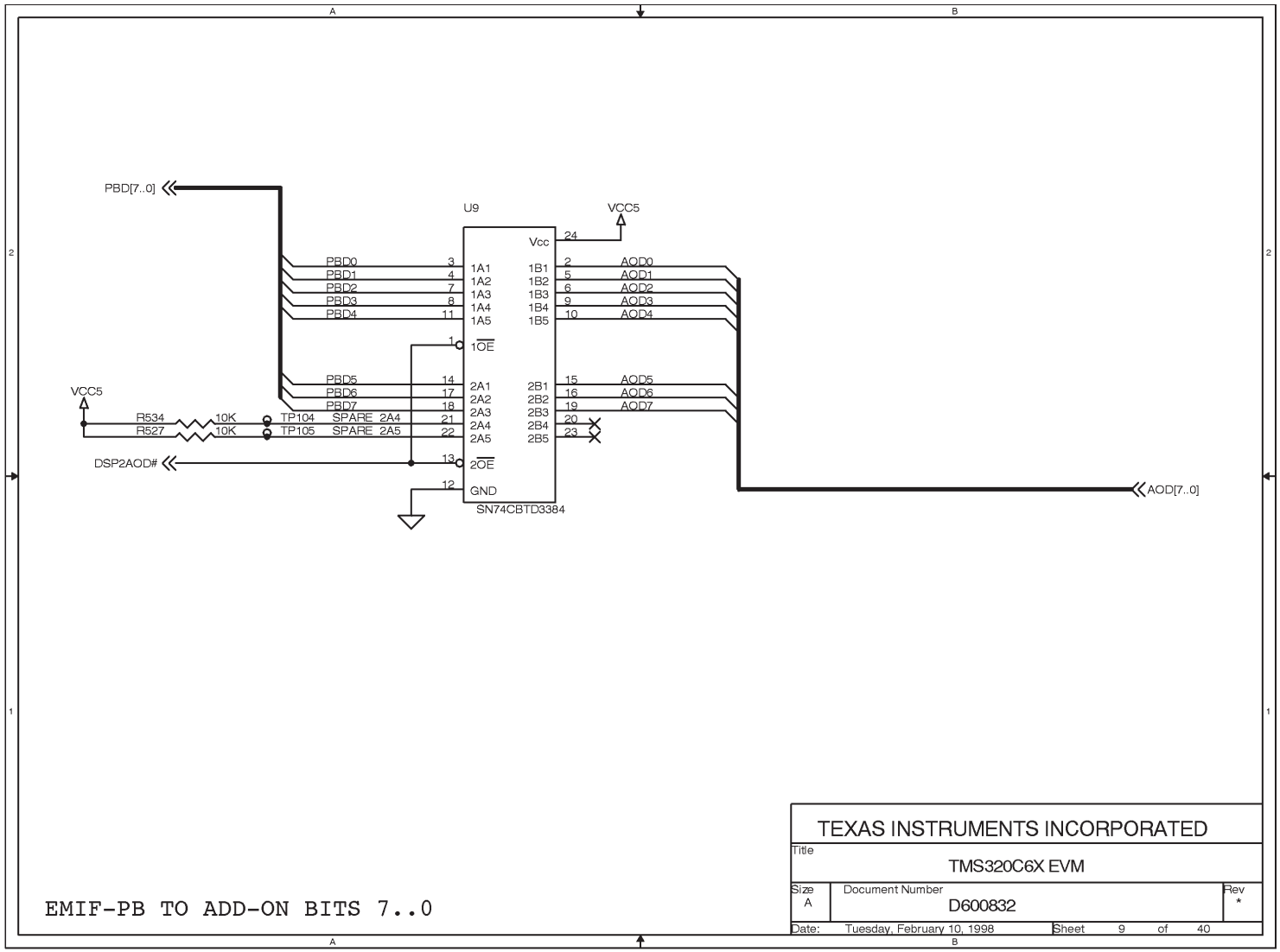

TEXAS INSTRUMENTS INCORPORATED
TMS320C6X EVM
Title
Size A
Document Number D600832
Rev *
Date: Tuesday, February 10, 1998
Sheet 6 of 40
EMIF-BUS SWITCH
U20
QS34X245
U504
TC7SHU04F
D[31..0]
ED[31..0]
CE0#
CE0
TP703
TP704
TP_CE0#
R581
R582
NU
VCC5
VCC3
See notes on pg. 1

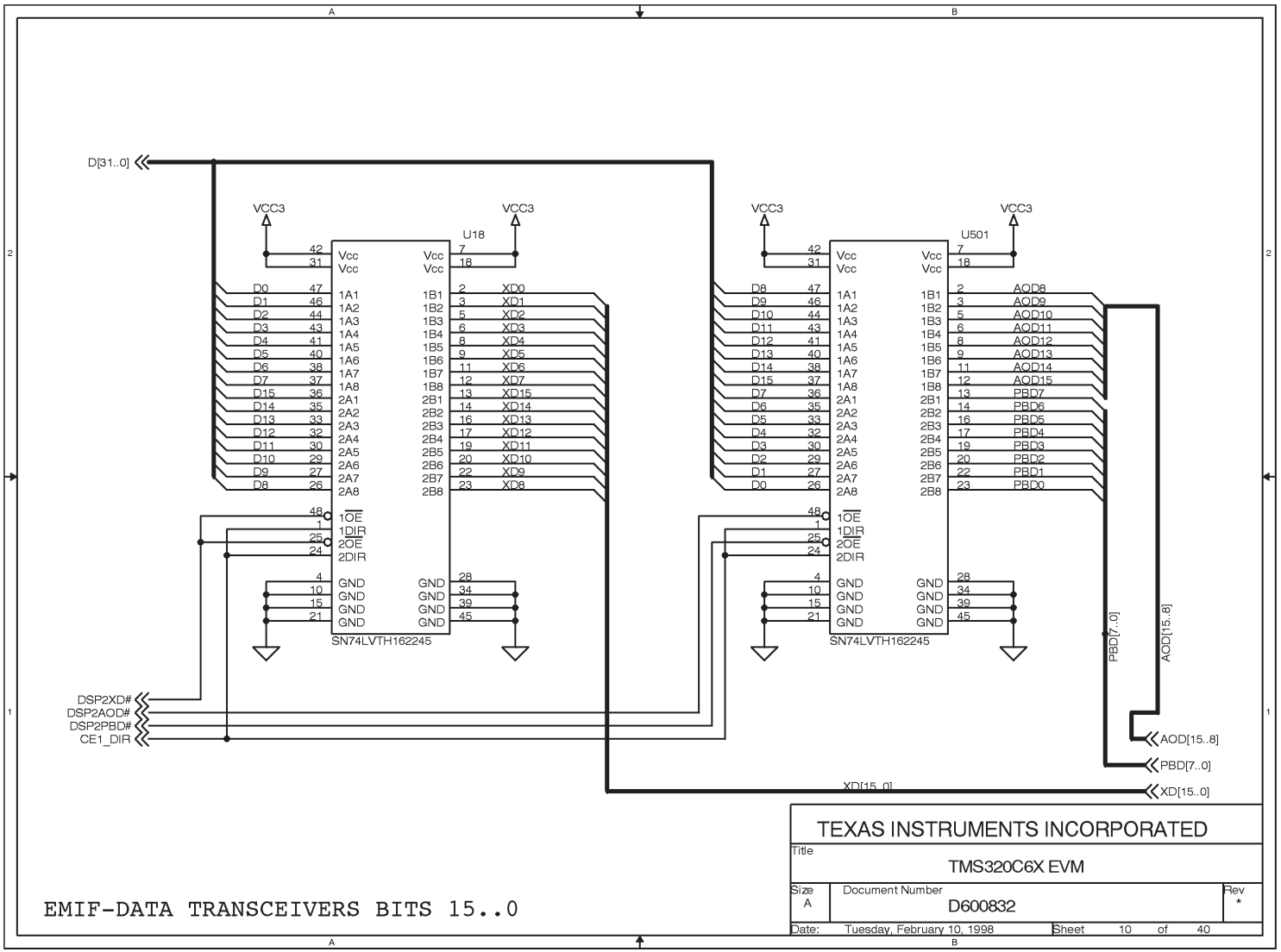


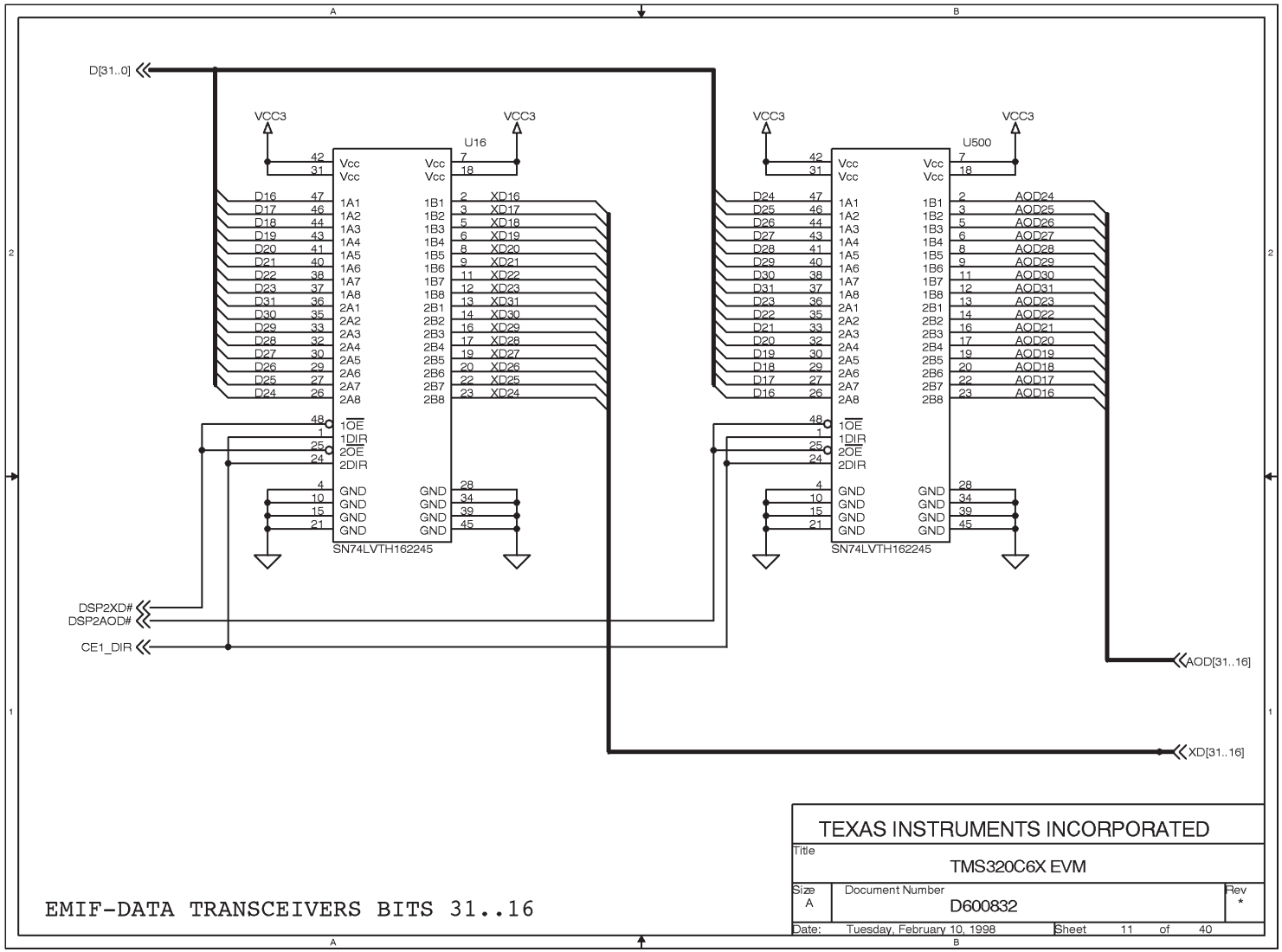
EMIF ADDRESS BUFFERS BITS 12..2

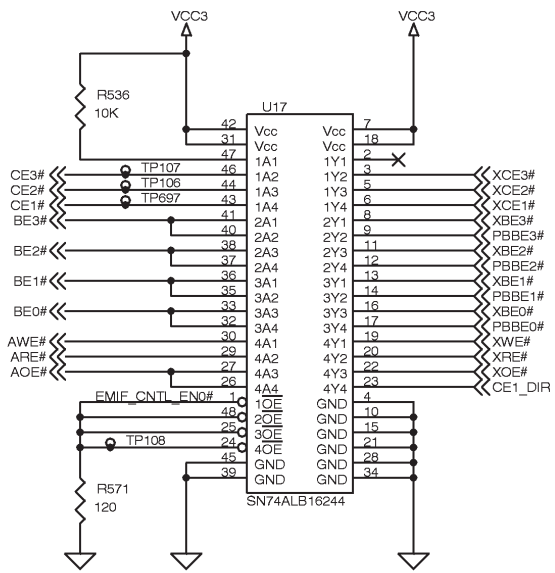
TEXAS INSTRUMENTS INCORPORATED		
Title: TMS320C6X EVM		
Size: A	Document Number: D600832	Rev: *
Date: Tuesday, February 10, 1998	Sheet 7 of 40	


EA[21..13]
VCC3
VCC3
U13
SN74ALB16244
EA13
EA14
EA15
EA16
EA17
EA18
EA19
EA20
EA21
EA21TO14
OE#
R544
120
TP103
SDA_T13
XA_T13
XA_T14
XA_T15
XA_T16
PBA16
XA_T17
PBA17
XA_T18
PBA18
XA_T19
PBA19
XA_T20
PBA20
XA_T21
PBA21
TP93
TP94
TP95
TP96
TP97
TP98
TP99
TP100
TP101
TP102
R24
R25
R26
R27
R28
R29
R30
R31
R32
R33
SDA13
XA13
XA14
XA15
XA16
XA17
XA18
XA19
XA20
XA21
XA[21..13]
PBA[21..16]
TEXAS INSTRUMENTS INCORPORATED
TMS320C6X EVM
D600832
Tuesday, February 10, 1998
EMIF-ADDRESS BUFFERS BITS 21..13


PBD[7..0]
U9
VCC5
Vcc
24
PBD0
PBD1
PBD2
PBD3
PBD4
1A1
1A2
1A3
1A4
1A5
1B1
1B2
1B3
1B4
1B5
AOD0
AOD1
AOD2
AOD3
AOD4
1OE
PBD5
PBD6
PBD7
2A1
2A2
2A3
2A4
2A5
2B1
2B2
2B3
2B4
2B5
AOD5
AOD6
AOD7
VCC5
R534
R527
10K
10K
TP104
TP105
SPARE 2A4
SPARE 2A5
DSP2AOD#
2OE
GND
SN74CBTD3384
AOD[7..0]
TEXAS INSTRUMENTS INCORPORATED
Title
TMS320C6X EVM
Size
A
Document Number
D600832
Rev
*
Date:
Tuesday, February 10, 1998
Sheet
9
of
40
EMIF-PB TO ADD-ON BITS 7..0

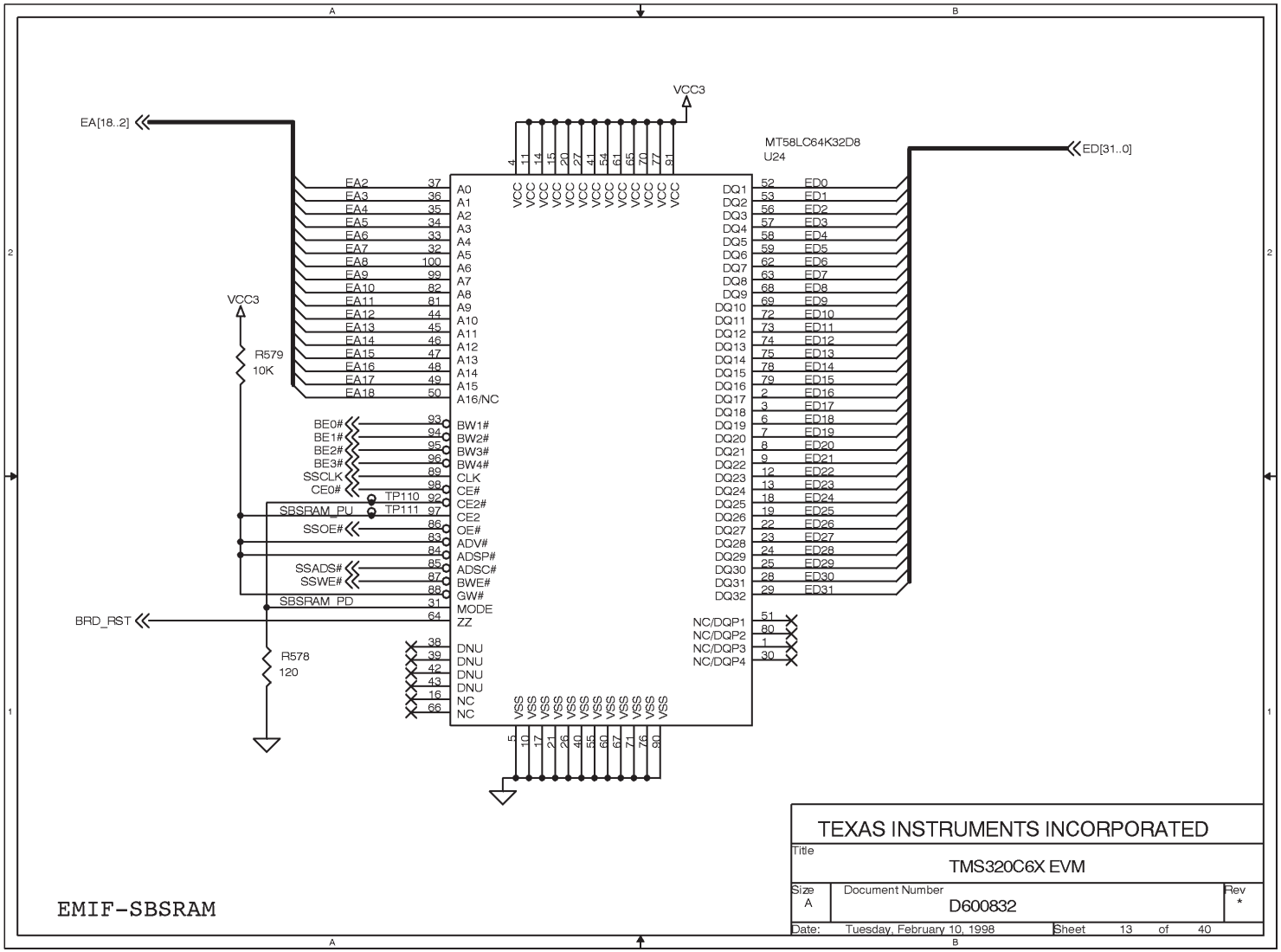

D[31..0]
VCC3
U18
U501
SN74LVTH162245
DSP2XD#
DSP2AOD#
DSP2PBD#
CE1_DIR
PBD[7..0]
AOD[15..8]
AOD[15..8]
PBD[7..0]
XD[15..0]
XD[15..0]
TEXAS INSTRUMENTS INCORPORATED
Title
TMS320C6X EVM
Size
A
Document Number
D600832
Rev
*
Date: Tuesday, February 10, 1998
Sheet 10 of 40
EMIF-DATA TRANSCEIVERS BITS 15..0

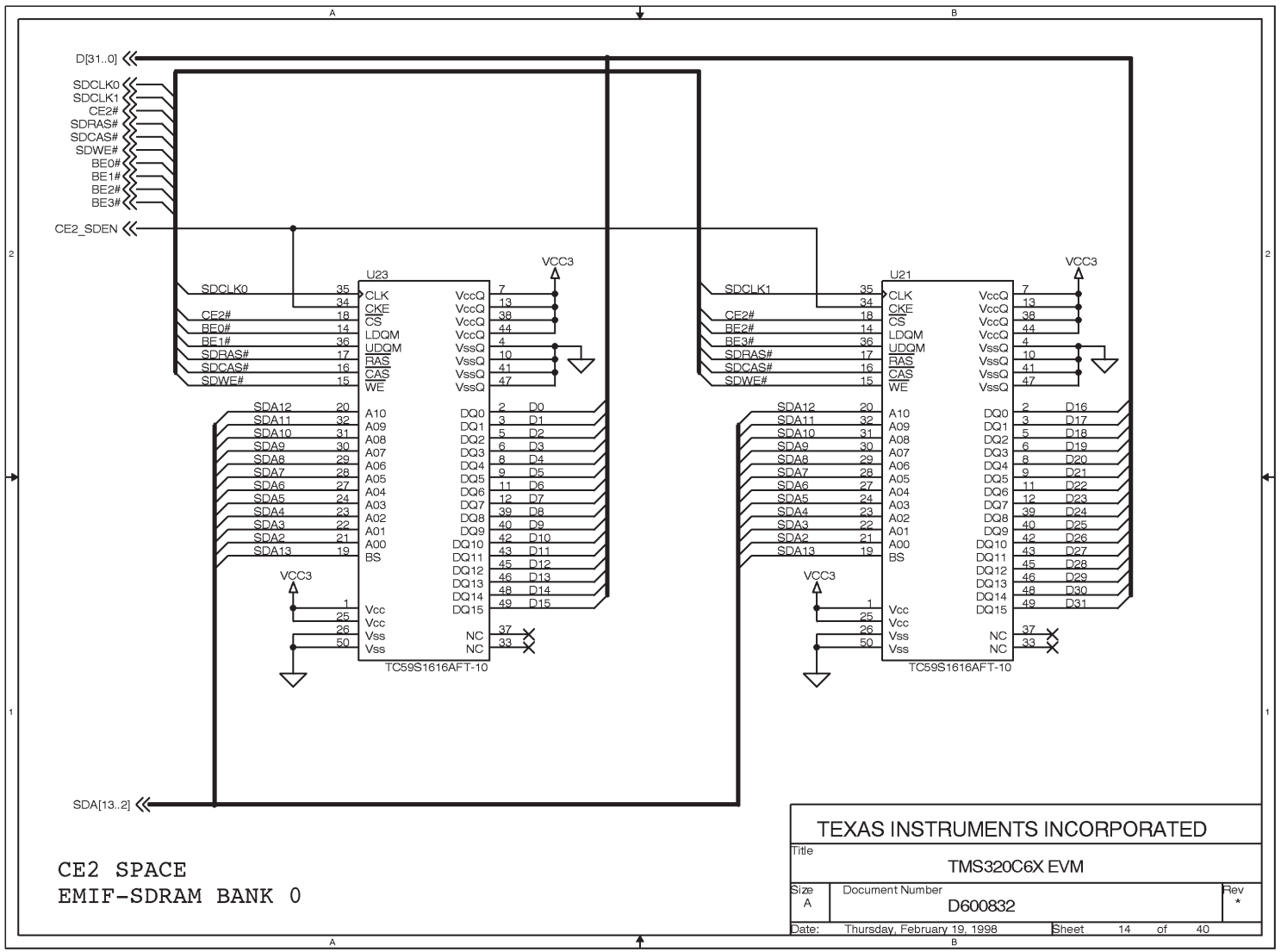

D[31..0]
VCC3
VCC3
U16
Vcc
Vcc
Vcc
Vcc
D16 D17 D18 D19 D20 D21 D22 D23 D31 D30 D29 D28 D27 D26 D25 D24
1A1 1A2 1A3 1A4 1A5 1A6 1A7 1A8 2A1 2A2 2A3 2A4 2A5 2A6 2A7 2A8
1B1 1B2 1B3 1B4 1B5 1B6 1B7 1B8 2B1 2B2 2B3 2B4 2B5 2B6 2B7 2B8
XD16 XD17 XD18 XD19 XD20 XD21 XD22 XD23 XD31 XD30 XD29 XD28 XD27 XD26 XD25 XD24
1OE
1DIR
2OE
2DIR
GND
SN74LVTH162245
VCC3
VCC3
U500
D24 D25 D26 D27 D28 D29 D30 D31 D23 D22 D21 D20 D19 D18 D17 D16
AOD24 AOD25 AOD26 AOD27 AOD28 AOD29 AOD30 AOD31 AOD23 AOD22 AOD21 AOD20 AOD19 AOD18 AOD17 AOD16
SN74LVTH162245
DSP2XD#
DSP2AOD#
CE1_DIR
AOD[31..16]
XD[31..16]
EMIF-DATA TRANSCEIVERS BITS 31..16
TEXAS INSTRUMENTS INCORPORATED
Title
TMS320C6X EVM
Size
A
Document Number
D600832
Rev
*
Date: Tuesday, February 10, 1998
Sheet 11 of 40

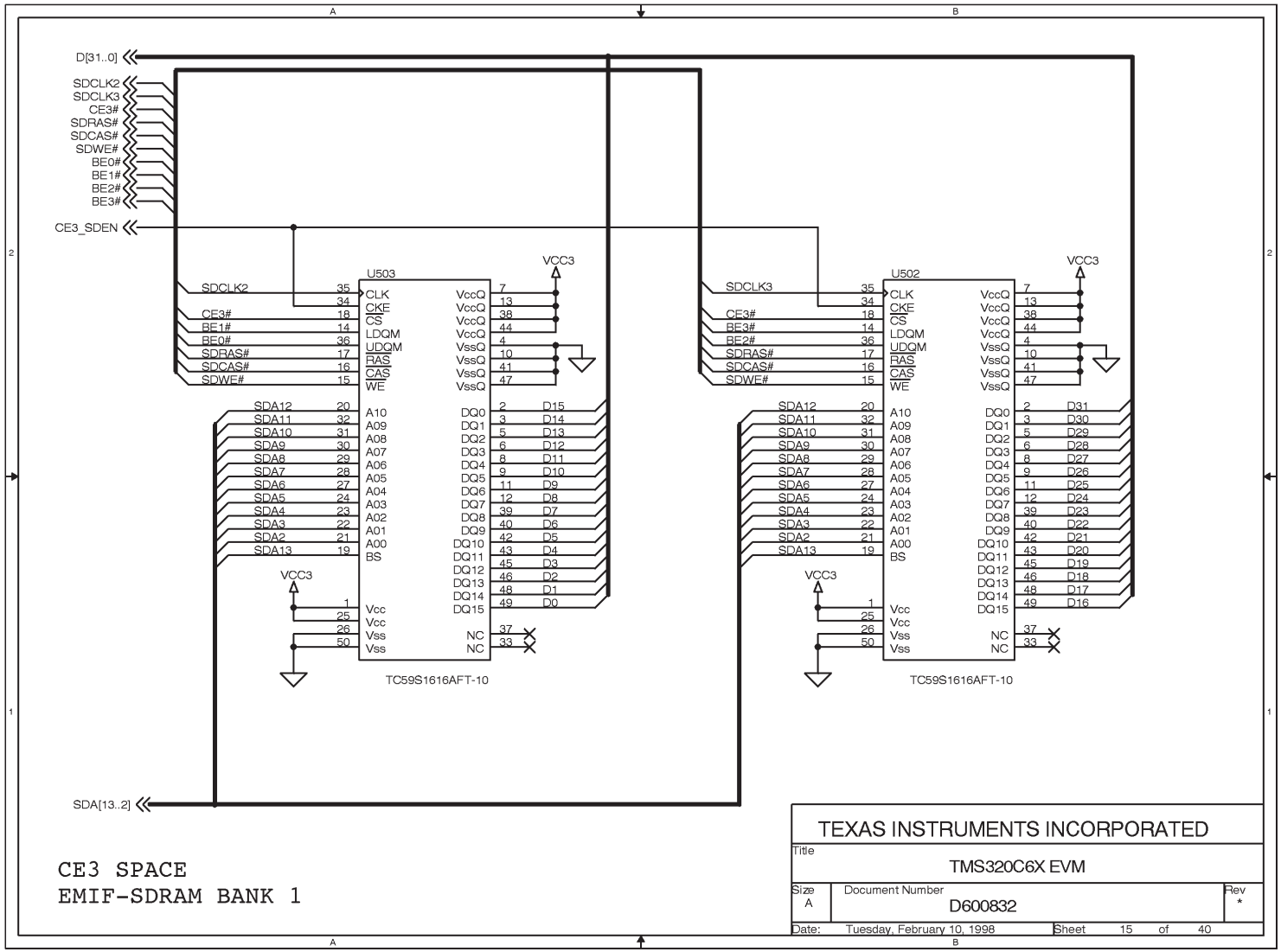


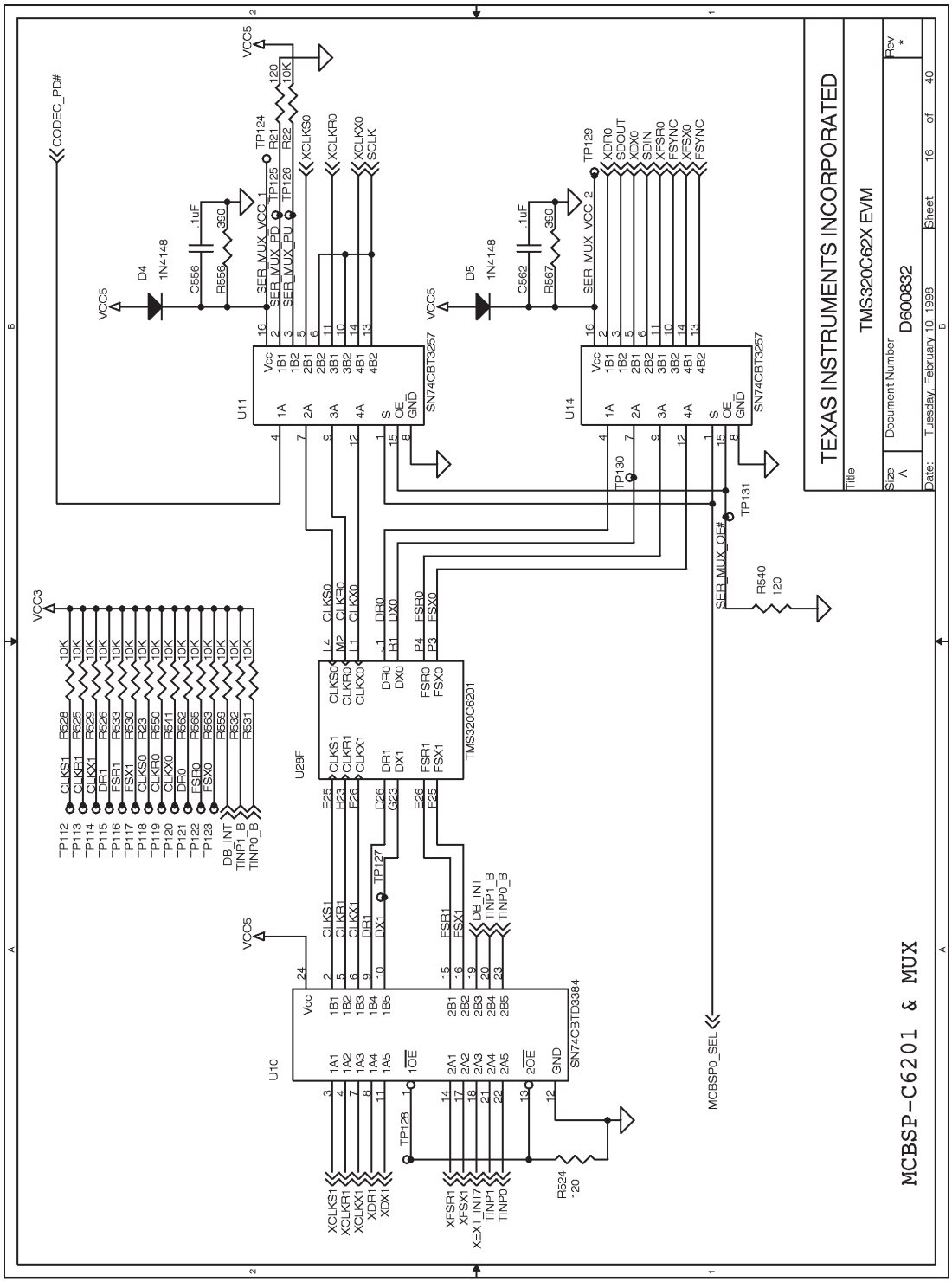
EMIF-CONTROL BUFFERS

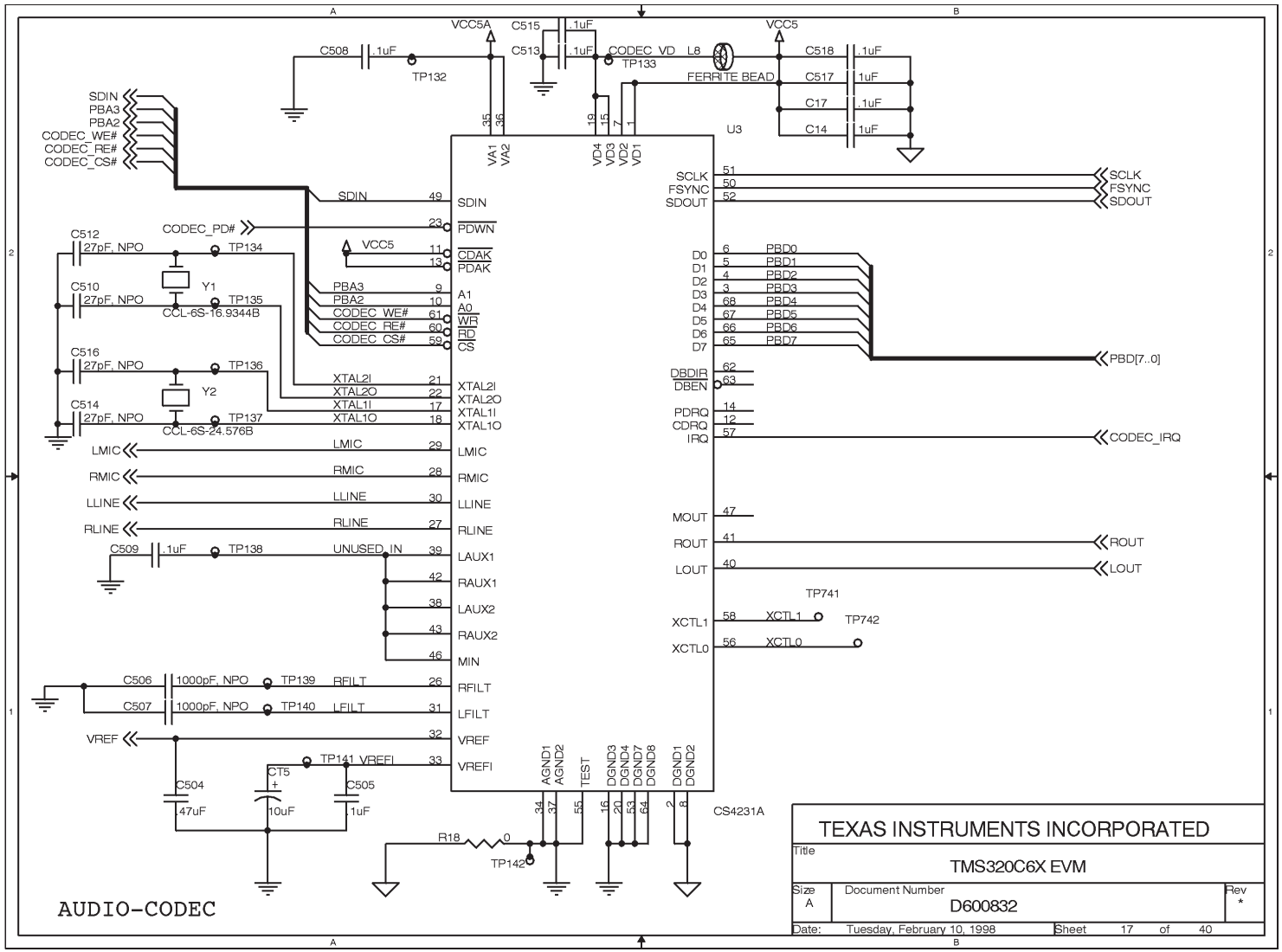
TEXAS INSTRUMENTS INCORPORATED		
Title: TMS320C6X EVM		
Size: A	Document Number: D600832	Rev: *
Date: Tuesday, February 10, 1998	Sheet 12 of 40	

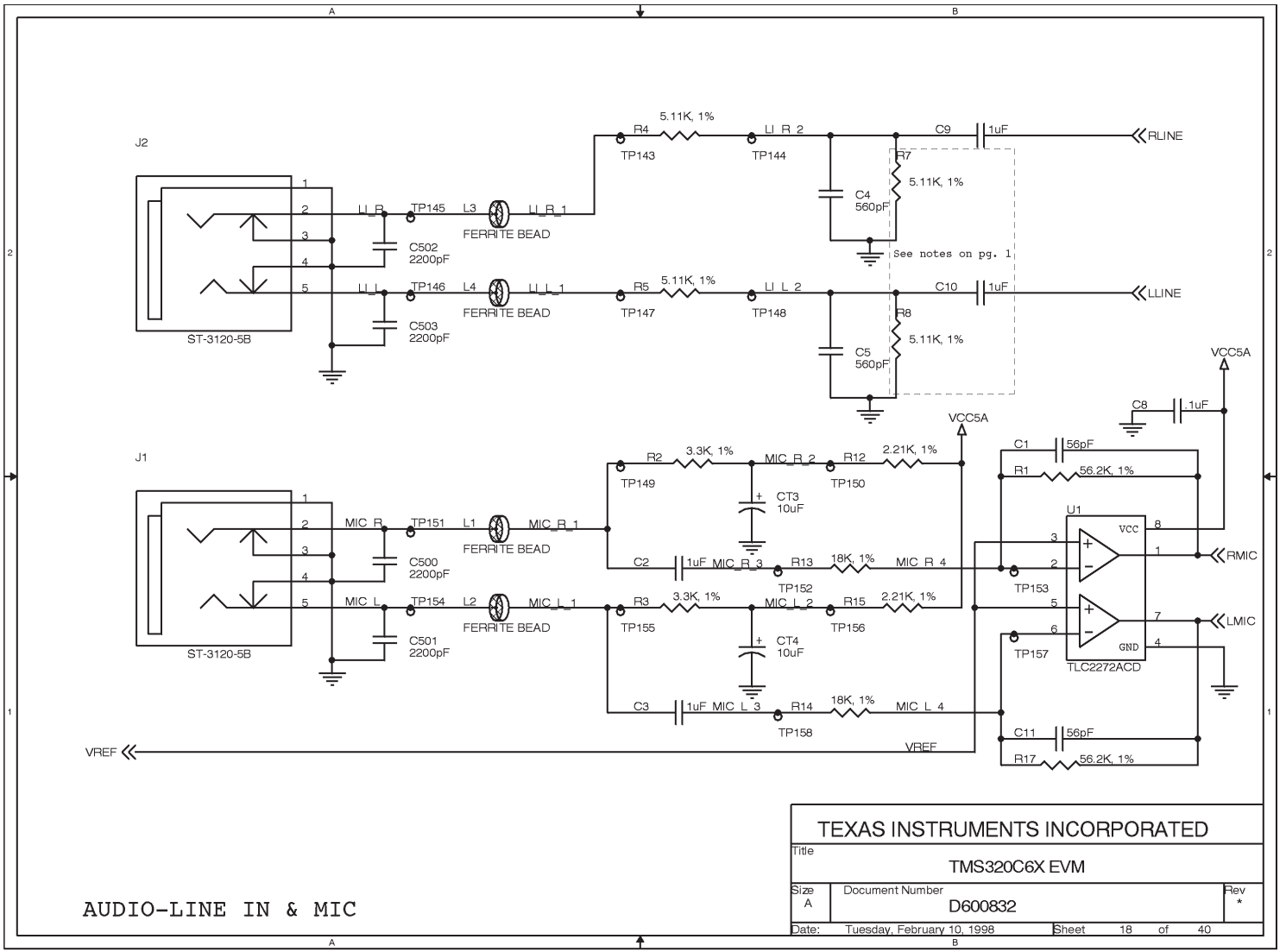

EA[18..2]
VCC3
MT58LC64K32D8
U24
ED[31..0]
R579
10K
R578
120
BRD_RST
SBSRAM_PU
SBSRAM_PD
TP110
TP111
BE0#
BE1#
BE2#
BE3#
SSCLK
CE0#
SSOE#
SSADS#
SSWE#
A0
A1
A2
A3
A4
A5
A6
A7
A8
A9
A10
A11
A12
A13
A14
A15
A16/NC
BW1#
BW2#
BW3#
BW4#
CLK
CE#
CE2#
CE2
OE#
ADV#
ADSP#
ADSC#
BWE#
GW#
MODE
ZZ
DNU
NC
DQ1
DQ2
DQ3
DQ4
DQ5
DQ6
DQ7
DQ8
DQ9
DQ10
DQ11
DQ12
DQ13
DQ14
DQ15
DQ16
DQ17
DQ18
DQ19
DQ20
DQ21
DQ22
DQ23
DQ24
DQ25
DQ26
DQ27
DQ28
DQ29
DQ30
DQ31
DQ32
NC/DQP1
NC/DQP2
NC/DQP3
NC/DQP4
VCC
VSS
EMIF-SBSRAM
TEXAS INSTRUMENTS INCORPORATED
Title
TMS320C6X EVM
Size
A
Document Number
D600832
Rev
*
Date: Tuesday, February 10, 1998
Sheet 13 of 40


D[31..0]
SDCLK0
SDCLK1
CE2#
SDRAS#
SDCAS#
SDWE#
BE0#
BE1#
BE2#
BE3#
CE2_SDEN
U23
U21
VCC3
CLK
CKE
CS
LDQM
UDQM
RAS
CAS
WE
A10
A09
A08
A07
A06
A05
A04
A03
A02
A01
A00
BS
Vcc
Vss
VccQ
VssQ
NC
DQ0
DQ1
DQ2
DQ3
DQ4
DQ5
DQ6
DQ7
DQ8
DQ9
DQ10
DQ11
DQ12
DQ13
DQ14
DQ15
TC59S1616AFT-10
SDA[13..2]
CE2 SPACE
EMIF-SDRAM BANK 0
TEXAS INSTRUMENTS INCORPORATED
Title
TMS320C6X EVM
Size
A
Document Number
D600832
Rev
*
Date: Thursday, February 19, 1998
Sheet 14 of 40


D[31..0]
SDCLK2
SDCLK3
CE3#
SDRAS#
SDCAS#
SDWE#
BE0#
BE1#
BE2#
BE3#
CE3_SDEN
U503
U502
VCC3
CLK
CKE
CS
LDQM
UDQM
RAS
CAS
WE
A10
A09
A08
A07
A06
A05
A04
A03
A02
A01
A00
BS
DQ0
DQ1
DQ2
DQ3
DQ4
DQ5
DQ6
DQ7
DQ8
DQ9
DQ10
DQ11
DQ12
DQ13
DQ14
DQ15
VccQ
VssQ
Vcc
Vss
NC
TC59S1616AFT-10
SDA[13..2]
CE3 SPACE
EMIF-SDRAM BANK 1
TEXAS INSTRUMENTS INCORPORATED
Title
TMS320C6X EVM
Size
A
Document Number
D600832
Rev
*
Date: Tuesday, February 10, 1998
Sheet 15 of 40

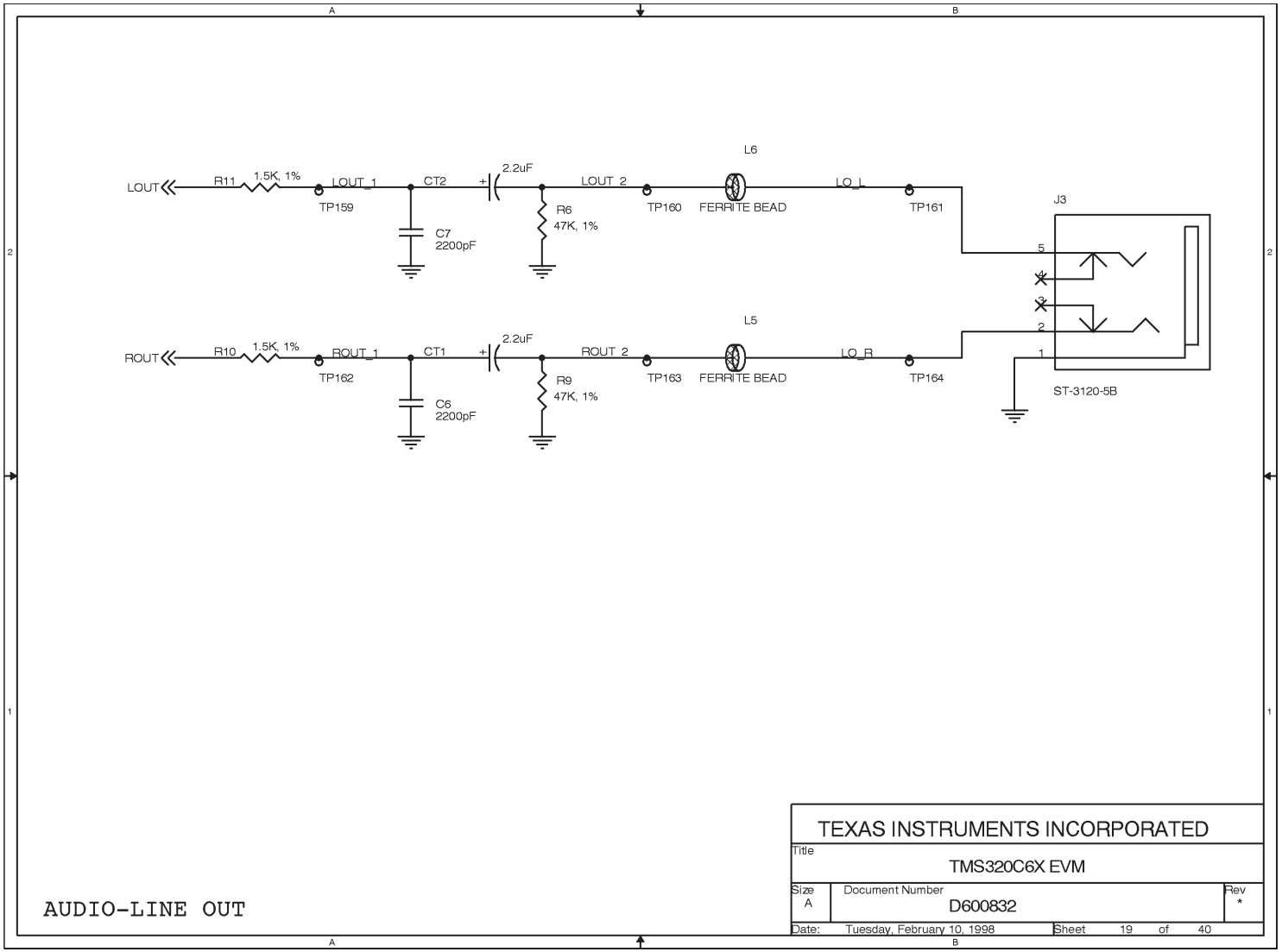

MCBSP-C6201 & MUX
TEXAS INSTRUMENTS INCORPORATED
TMS320C62X EVM
D600832
Rev *
Tuesday, February 10, 1998
Sheet 16 of 40
CODEC_PD#
SER_MUX_VCC_1
SER_MUX_PD
SER_MUX_PU
SER_MUX_VCC_2
SER_MUX_OE#
MCBSP0_SEL
XCLKS0
XCLKR0
XCLKX0
SCLK
XDR0
SDOUT
XDX0
SDIN
XFSR0
FSYNC
XFSX0
FSYNC
U11
U14
U10
U28F
SN74CBT3257
SN74CBTD3384
TMS320C6201
D4
D5
1N4148
C556
C562
R556
R567
1uF
390
120
10K
R524
R540
R21
R22
TP124
TP125
TP126
TP129
TP130
TP131
TP127
TP128
VCC5
VCC3
XCLKS1
XCLKR1
XCLKX1
XDR1
XDX1
XFSR1
XFSX1
XEXT_INT7
TINP1
TINP0
DB_INT
TINP1_B
TINP0_B

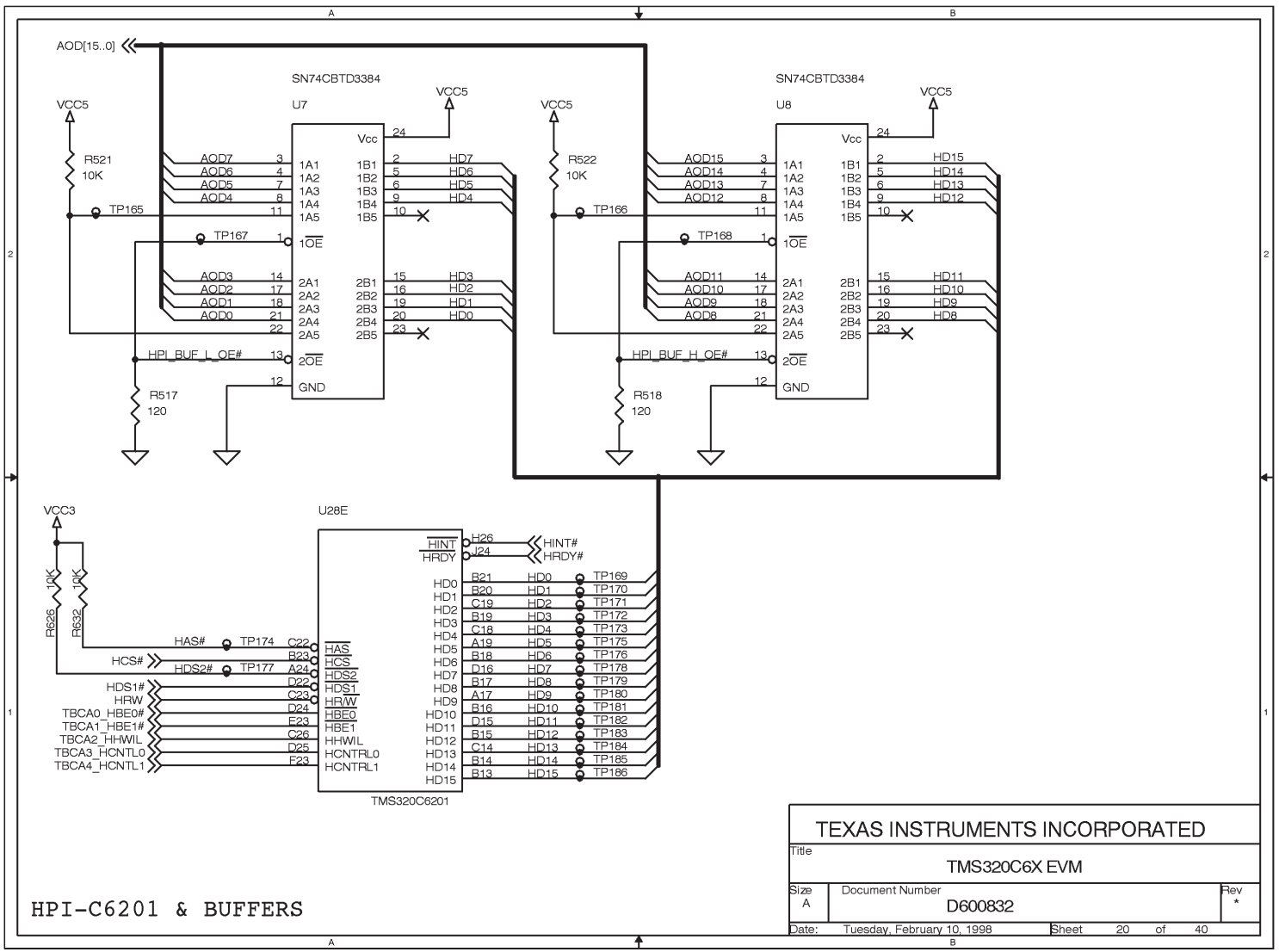

AUDIO-CODEC
TEXAS INSTRUMENTS INCORPORATED
Title
TMS320C6X EVM
Size A
Document Number D600832
Rev *
Date: Tuesday, February 10, 1998
Sheet 17 of 40
CS4231A
U3
VCC5A
VCC5
C508 .1uF
C515 .1uF
C513 .1uF
CODEC_VD
L8
FERRITE BEAD
C518 .1uF
C517 1uF
C17 .1uF
C14 1uF
TP132
TP133
SDIN
PBA3
PBA2
CODEC_WE#
CODEC_RE#
CODEC_CS#
CODEC_PD#
C512 27pF, NPO
C510 27pF, NPO
C516 27pF, NPO
C514 27pF, NPO
Y1
Y2
CCL-6S-16.9344B
CCL-6S-24.576B
TP134
TP135
TP136
TP137
C509 .1uF
TP138
UNUSED_IN
LMIC
RMIC
LLINE
RLINE
C506 1000pF, NPO
C507 1000pF, NPO
TP139
TP140
RFILT
LFILT
VREF
TP141 VREFI
C504 .47uF
CT5 10uF
C505 .1uF
R18 0
TP142
SCLK
FSYNC
SDOUT
PBD0
PBD1
PBD2
PBD3
PBD4
PBD5
PBD6
PBD7
PBD[7..0]
DBDIR
DBEN
PDRQ
CDRQ
IRQ
CODEC_IRQ
MOUT
ROUT
LOUT
TP741
TP742
XCTL1
XCTL0
XTAL2I
XTAL2O
XTAL1I
XTAL1O
PDWN
CDAK
PDAK
A1
A0
WR
RD
CS
LAUX1
RAUX1
LAUX2
RAUX2
MIN
VA1
VA2
VD4
VD3
VD2
VD1
AGND1
AGND2
TEST
DGND3
DGND4
DGND7
DGND8
DGND1
DGND2

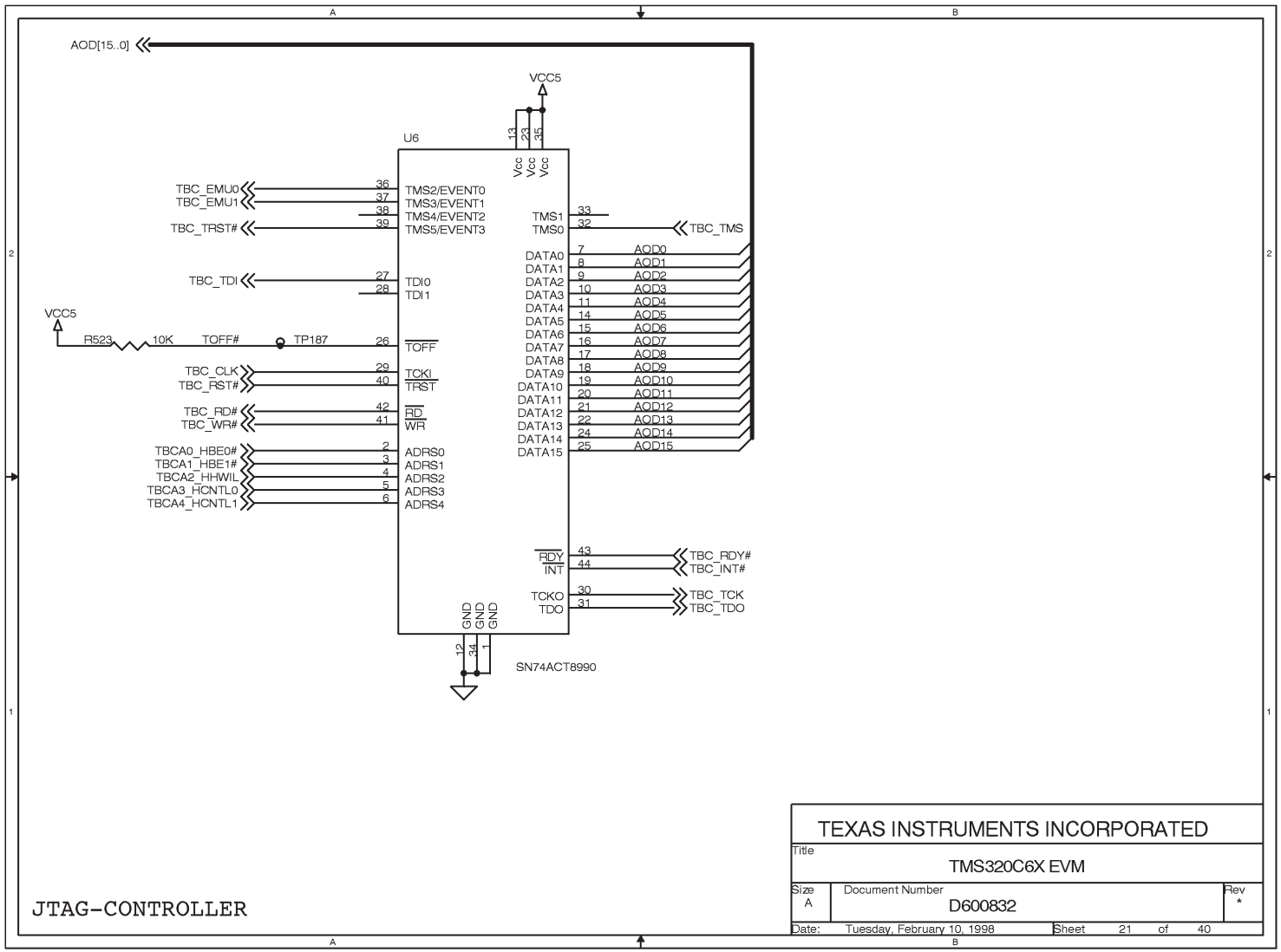


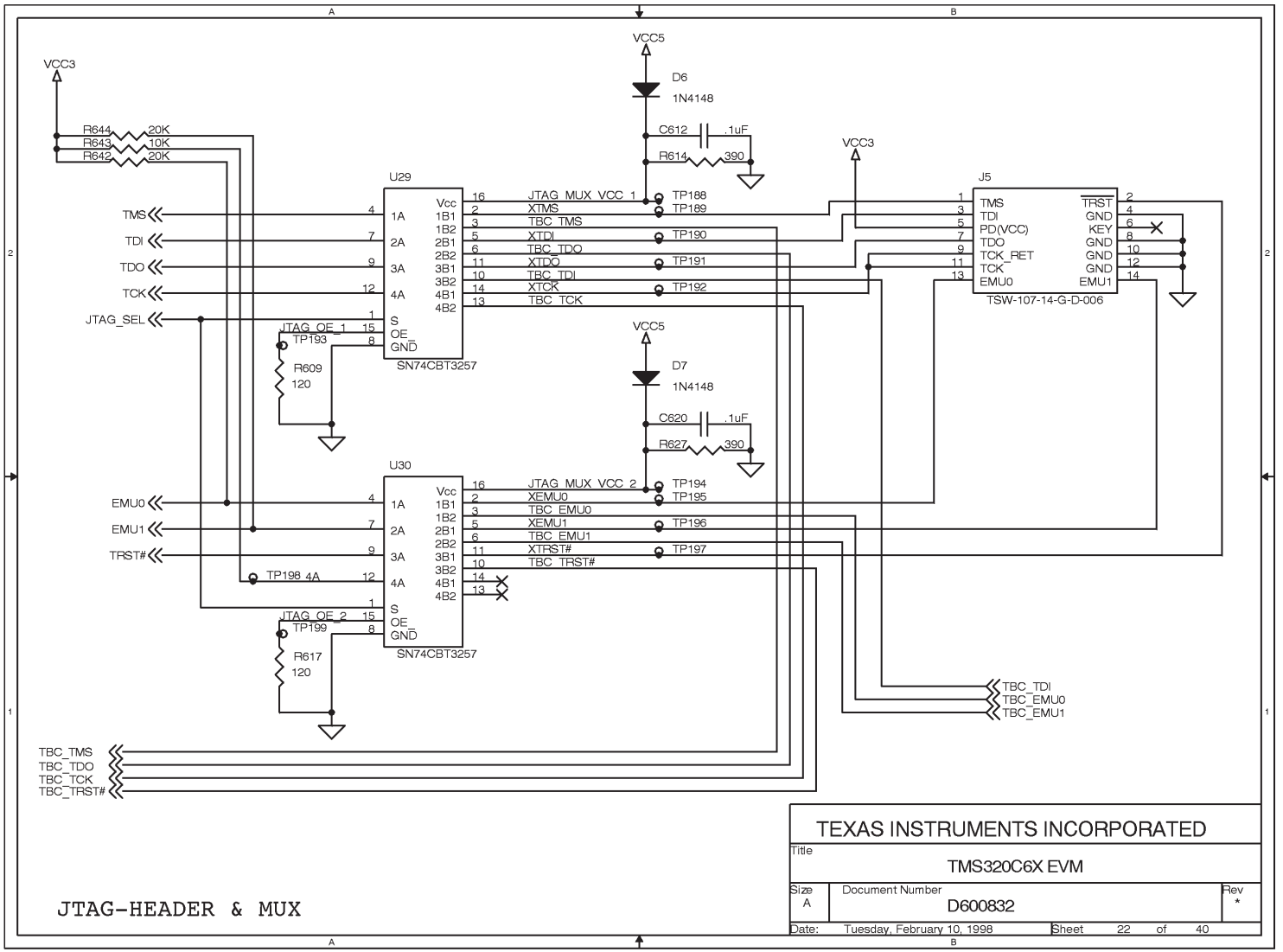
AUDIO-LINE IN & MIC

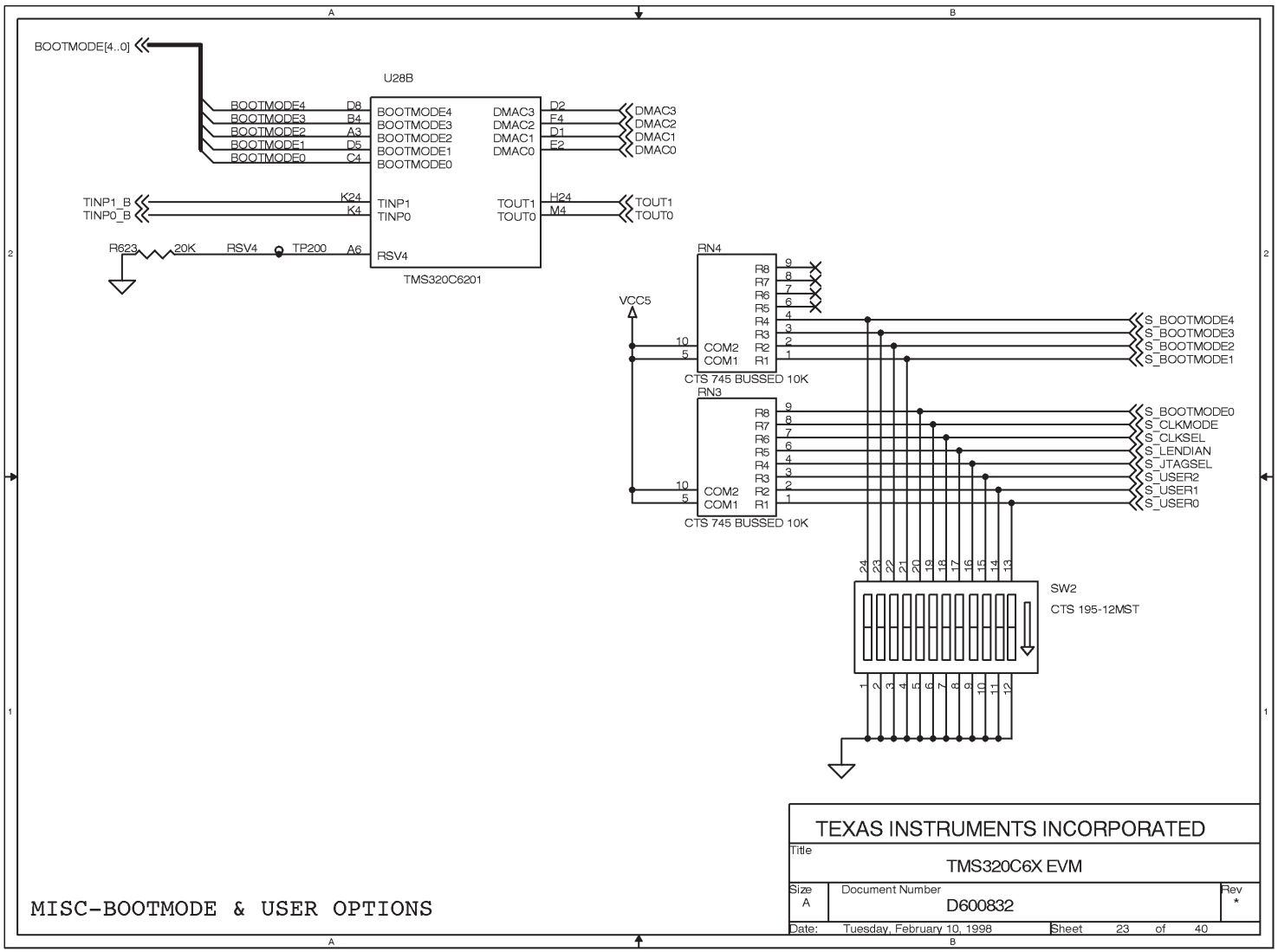
TEXAS INSTRUMENTS INCORPORATED		
Title: TMS320C6X EVM		
Size: A	Document Number: D600832	Rev: *
Date: Tuesday, February 10, 1998	Sheet 18 of 40	

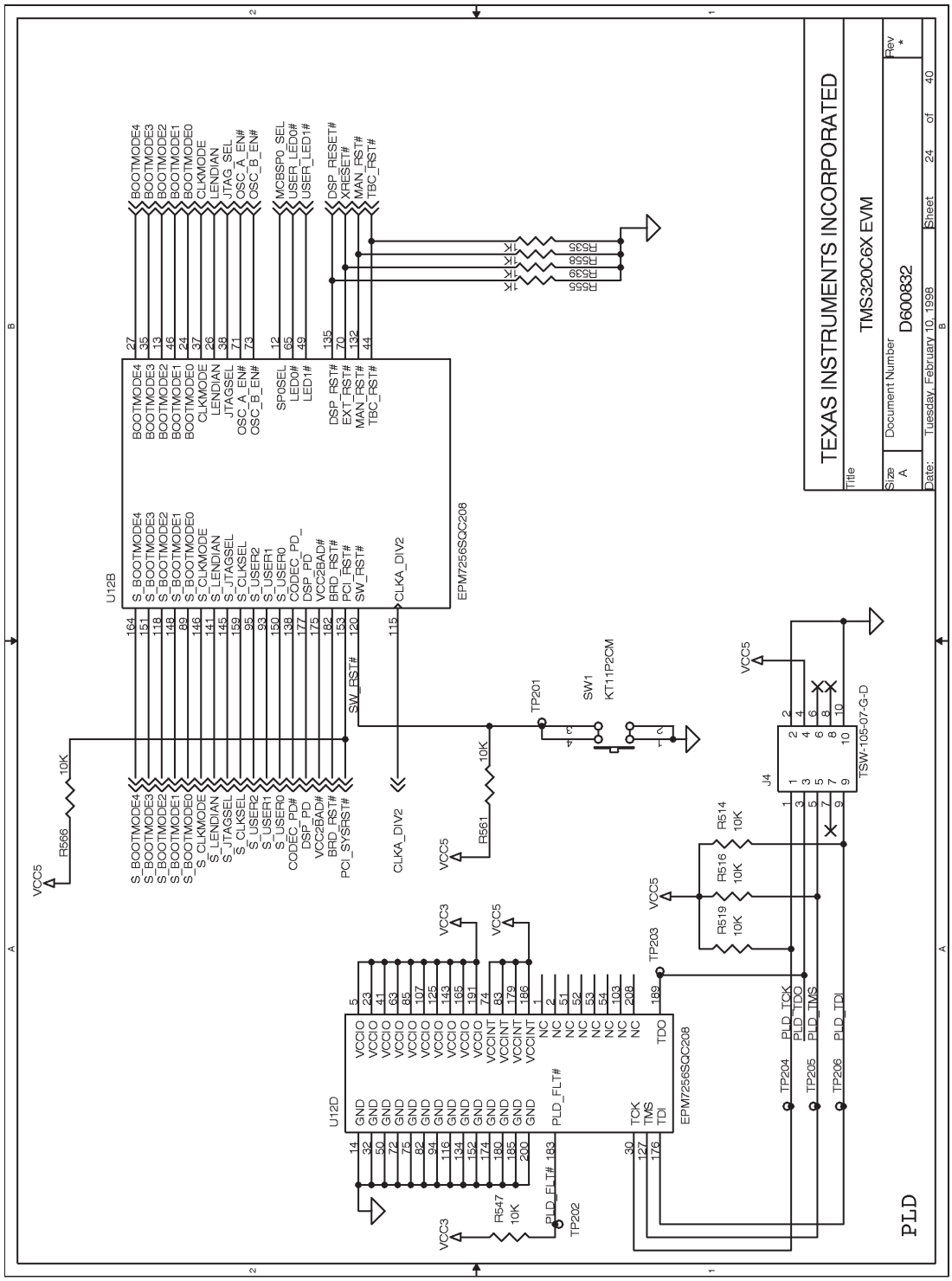

LOUT
R11
1.5K, 1%
LOUT_1
TP159
CT2
2.2uF
C7
2200pF
R6
47K, 1%
LOUT_2
TP160
L6
FERRITE BEAD
LO_L
TP161
ROUT
R10
1.5K, 1%
ROUT_1
TP162
CT1
2.2uF
C6
2200pF
R9
47K, 1%
ROUT_2
TP163
L5
FERRITE BEAD
LO_R
TP164
J3
5
4
3
2
1
ST-3120-5B
AUDIO-LINE OUT
TEXAS INSTRUMENTS INCORPORATED
Title
TMS320C6X EVM
Size
A
Document Number
D600832
Rev
*
Date:
Tuesday, February 10, 1998
Sheet
19
of
40

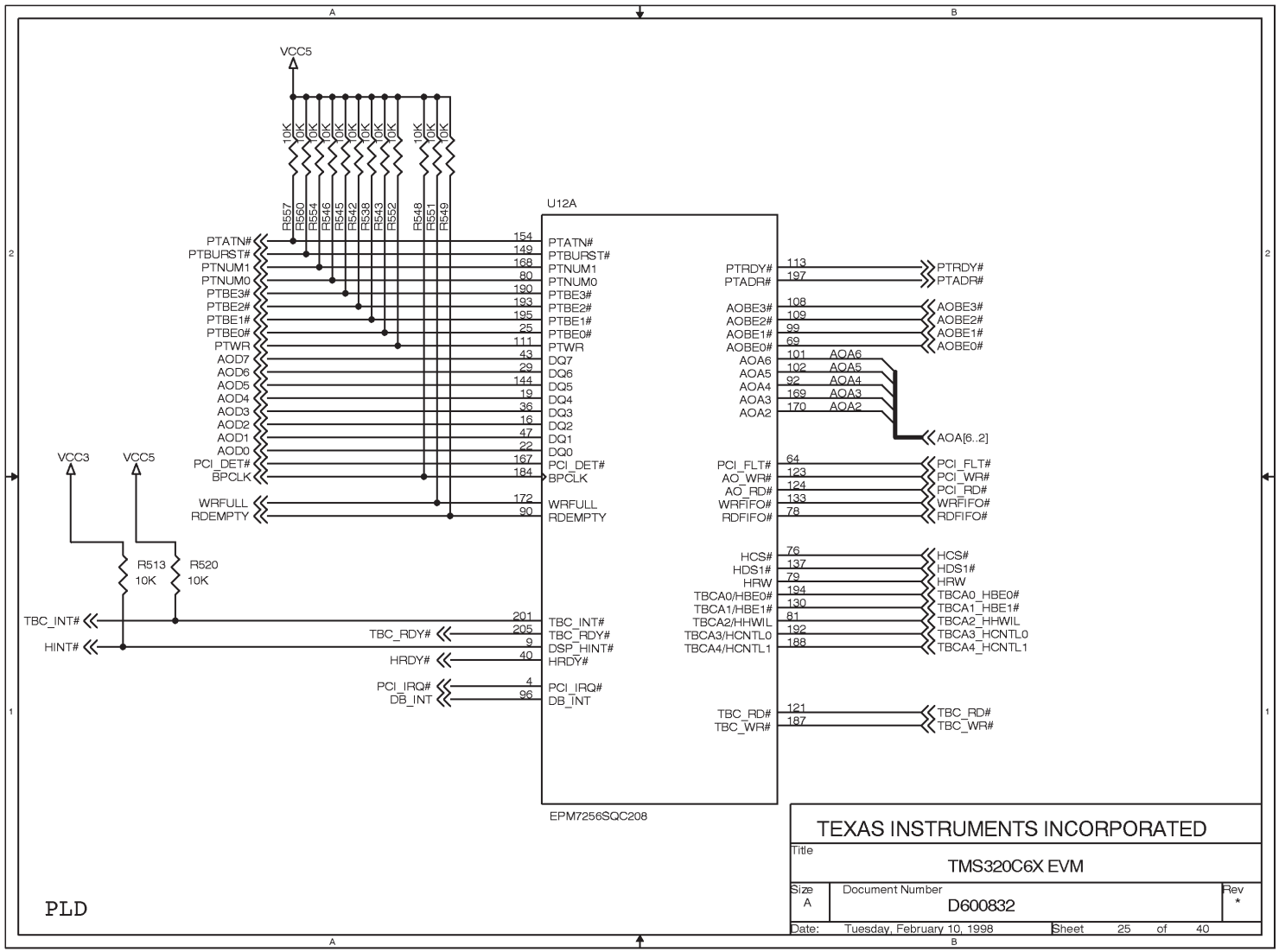

AOD[15..0]
SN74CBTD3384
U7
SN74CBTD3384
U8
VCC5
R521
10K
R522
10K
TP165
TP166
TP167
TP168
HPI_BUF_L_OE#
HPI_BUF_H_OE#
R517
120
R518
120
VCC3
U28E
R626
10K
R632
10K
HAS#
HCS#
HDS2#
TP174
TP177
HDS1#
HRW
TBCA0_HBE0#
TBCA1_HBE1#
TBCA2_HHWIL
TBCA3_HCNTL0
TBCA4_HCNTL1
HINT#
HRDY#
TMS320C6201
HPI-C6201 & BUFFERS
TEXAS INSTRUMENTS INCORPORATED
TMS320C6X EVM
D600832
Tuesday, February 10, 1998
Sheet 20 of 40

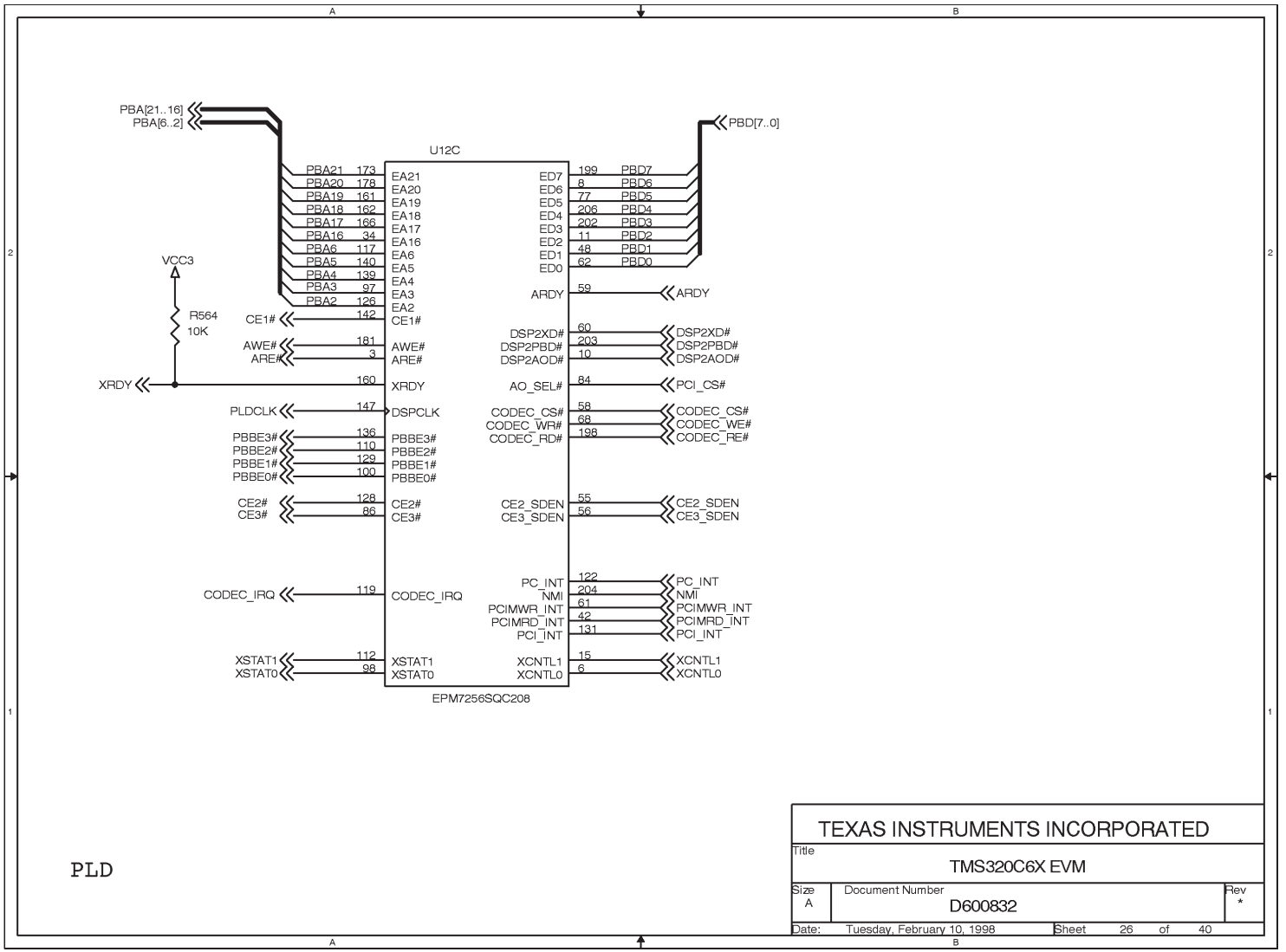

AOD[15..0]
VCC5
U6
13
23
35
Vcc
Vcc
Vcc
TBC_EMU0
TBC_EMU1
TBC_TRST#
36
37
38
39
TMS2/EVENT0
TMS3/EVENT1
TMS4/EVENT2
TMS5/EVENT3
TMS1
TMS0
33
32
TBC_TMS
TBC_TDI
27
28
TDI0
TDI1
VCC5
R523
10K
TOFF#
TP187
26
TOFF
TBC_CLK
TBC_RST#
29
40
TCKI
TRST
TBC_RD#
TBC_WR#
42
41
RD
WR
TBCA0_HBE0#
TBCA1_HBE1#
TBCA2_HHWIL
TBCA3_HCNTL0
TBCA4_HCNTL1
2
3
4
5
6
ADRS0
ADRS1
ADRS2
ADRS3
ADRS4
DATA0
DATA1
DATA2
DATA3
DATA4
DATA5
DATA6
DATA7
DATA8
DATA9
DATA10
DATA11
DATA12
DATA13
DATA14
DATA15
7
8
9
10
11
14
15
16
17
18
19
20
21
22
24
25
AOD0
AOD1
AOD2
AOD3
AOD4
AOD5
AOD6
AOD7
AOD8
AOD9
AOD10
AOD11
AOD12
AOD13
AOD14
AOD15
RDY
INT
43
44
TBC_RDY#
TBC_INT#
TCKO
TDO
30
31
TBC_TCK
TBC_TDO
GND
GND
GND
12
34
1
SN74ACT8990
JTAG-CONTROLLER
TEXAS INSTRUMENTS INCORPORATED
Title
TMS320C6X EVM
Size
A
Document Number
D600832
Rev
*
Date:
Tuesday, February 10, 1998
Sheet
21
of
40

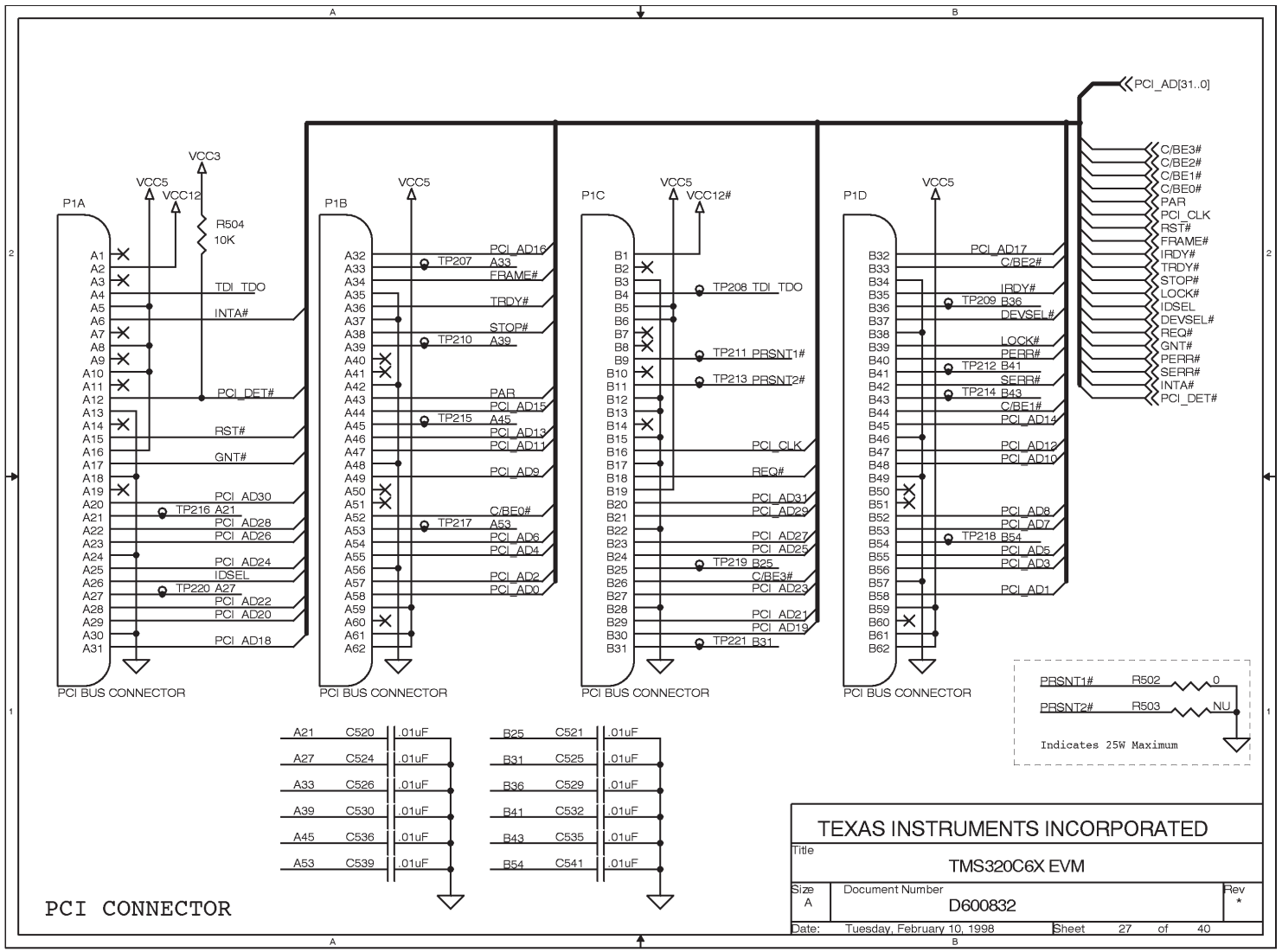

VCC3
VCC5
D6
1N4148
R644 20K
R643 10K
R642 20K
C612 .1uF
R614 390
U29
TMS
TDI
TDO
TCK
JTAG_SEL
JTAG_OE_1
TP193
R609
120
SN74CBT3257
JTAG_MUX_VCC_1
TP188
XTMS
TBC_TMS
TP189
XTDI
TBC_TDO
TP190
XTDO
TBC_TDI
TP191
XTCK
TBC_TCK
TP192
J5
TMS
TDI
PD(VCC)
TDO
TCK_RET
TCK
EMU0
TRST
GND
KEY
GND
GND
GND
EMU1
TSW-107-14-G-D-006
D7
1N4148
C620 .1uF
R627 390
U30
EMU0
EMU1
TRST#
TP198 4A
JTAG_OE_2
TP199
R617
120
SN74CBT3257
JTAG_MUX_VCC_2
TP194
XEMU0
TBC_EMU0
TP195
XEMU1
TBC_EMU1
TP196
XTRST#
TBC_TRST#
TP197
TBC_TDI
TBC_EMU0
TBC_EMU1
TBC_TMS
TBC_TDO
TBC_TCK
TBC_TRST#
JTAG-HEADER & MUX
TEXAS INSTRUMENTS INCORPORATED
Title
TMS320C6X EVM
Size
A
Document Number
D600832
Rev
*
Date: Tuesday, February 10, 1998
Sheet 22 of 40

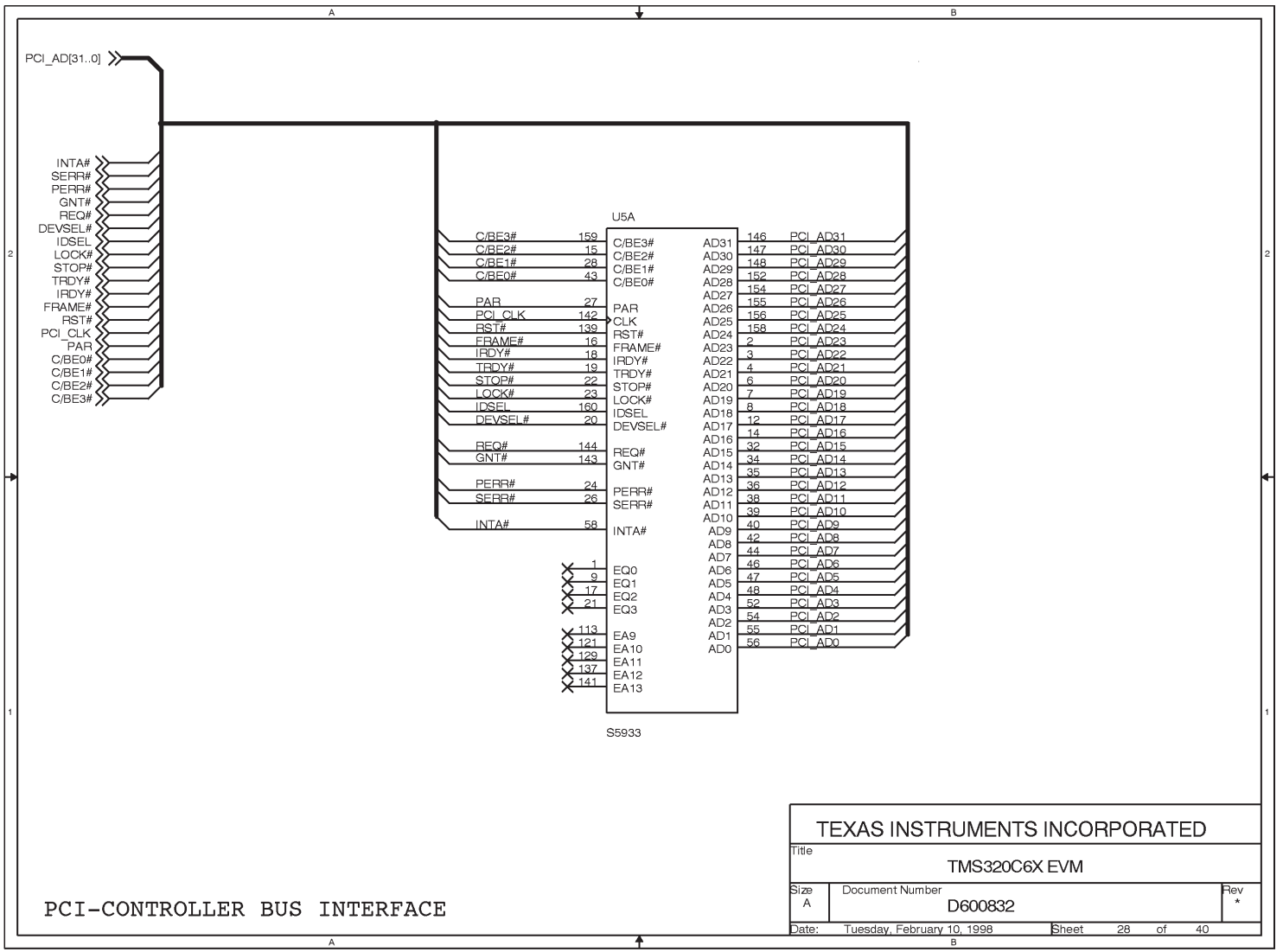

BOOTMODE[4..0]
U28B
BOOTMODE4 D8
BOOTMODE3 B4
BOOTMODE2 A3
BOOTMODE1 D5
BOOTMODE0 C4
BOOTMODE4
BOOTMODE3
BOOTMODE2
BOOTMODE1
BOOTMODE0
DMAC3 D2
DMAC2 F4
DMAC1 D1
DMAC0 E2
DMAC3
DMAC2
DMAC1
DMAC0
TINP1_B K24
TINP0_B K4
TINP1
TINP0
TOUT1 H24
TOUT0 M4
TOUT1
TOUT0
R623 20K RSV4 TP200 A6
RSV4
TMS320C6201
RN4
VCC5
10
5
COM2
COM1
R8 9
R7 8
R6 7
R5 6
R4 4
R3 3
R2 2
R1 1
CTS 745 BUSSED 10K
S_BOOTMODE4
S_BOOTMODE3
S_BOOTMODE2
S_BOOTMODE1
RN3
CTS 745 BUSSED 10K
S_BOOTMODE0
S_CLKMODE
S_CLKSEL
S_LENDIAN
S_JTAGSEL
S_USER2
S_USER1
S_USER0
SW2
CTS 195-12MST
TEXAS INSTRUMENTS INCORPORATED
Title
TMS320C6X EVM
Size A
Document Number
D600832
Rev *
Date: Tuesday, February 10, 1998
Sheet 23 of 40
MISC-BOOTMODE & USER OPTIONS

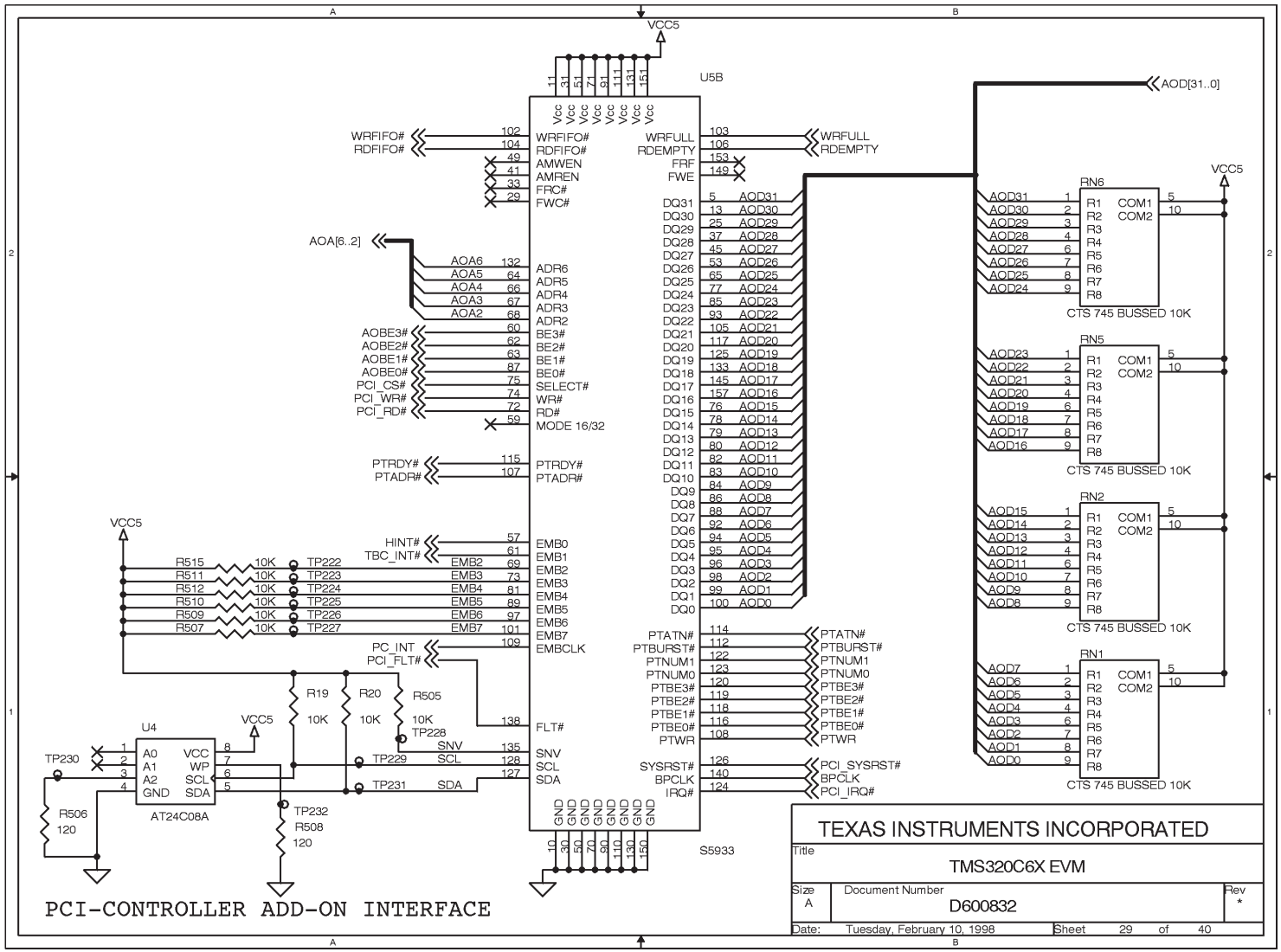

TEXAS INSTRUMENTS INCORPORATED
TMS320C6X EVM
Title
Size A
Document Number D600832
Rev *
Date: Tuesday, February 10, 1998
Sheet 24 of 40
PLD
U12B
EPM7256SQC208
U12D
EPM7256SQC208
J4
TSW-105-07-G-D
SW1
KT11P2CM
TP201
TP202
TP203
TP204
TP205
TP206
R566 10K
R561 10K
R547 10K
R514 10K
R516 10K
R519 10K
VCC5
VCC3
CLKA_DIV2
SW_RST#
PLD_FLT#
PLD_TCK
PLD_TDO
PLD_TMS
PLD_TDI

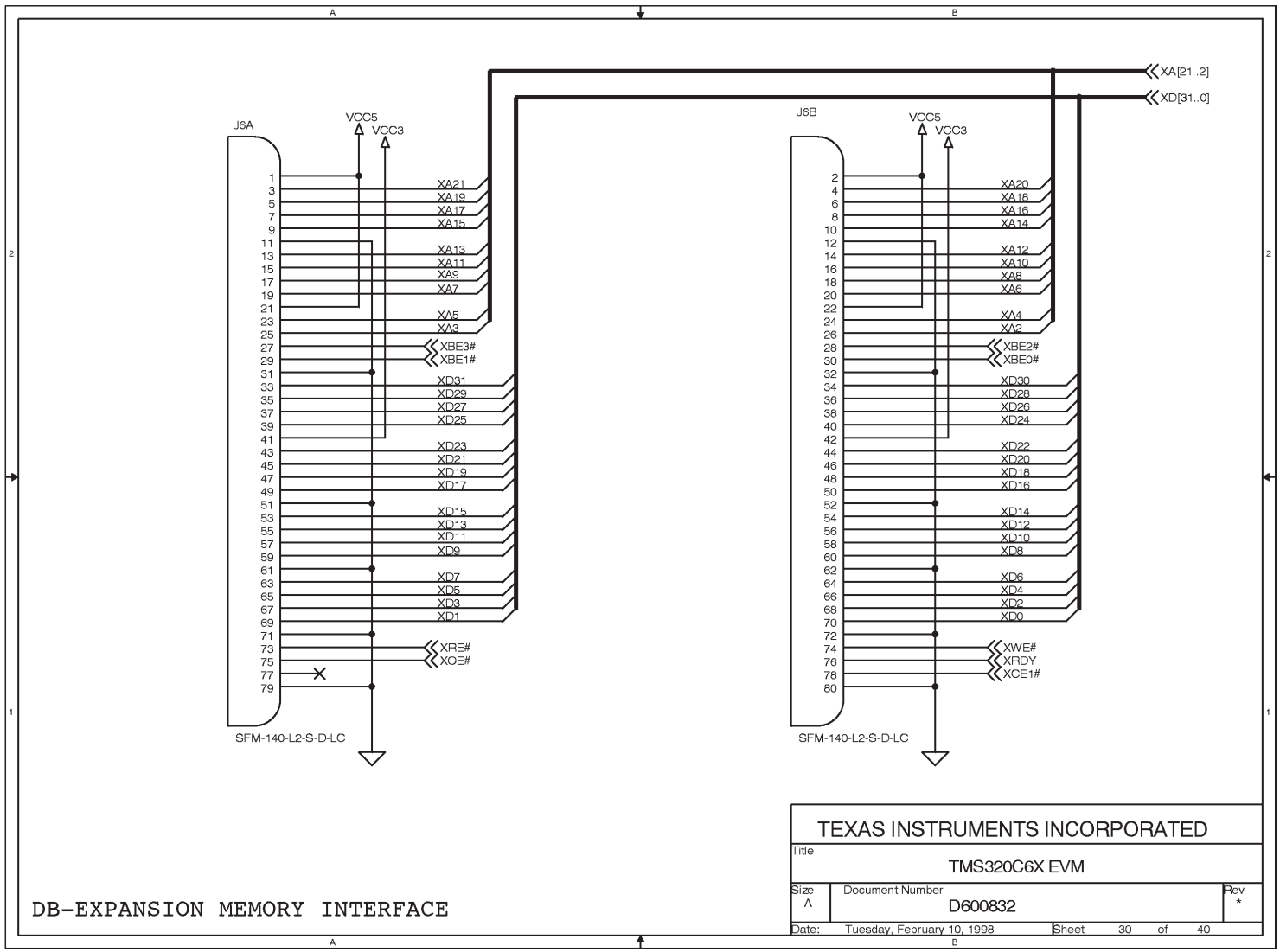

VCC5
10K
R557
R560
R554
R546
R545
R542
R538
R543
R552
R548
R551
R549
U12A
PTATN#
PTBURST#
PTNUM1
PTNUM0
PTBE3#
PTBE2#
PTBE1#
PTBE0#
PTWR
AOD7
AOD6
AOD5
AOD4
AOD3
AOD2
AOD1
AOD0
PCI_DET#
BPCLK
WRFULL
RDEMPTY
154
149
168
80
190
193
195
25
111
43
29
144
19
36
16
47
22
167
184
172
90
PTRDY#
PTADR#
AOBE3#
AOBE2#
AOBE1#
AOBE0#
AOA6
AOA5
AOA4
AOA3
AOA2
AOA[6..2]
PCI_FLT#
AO_WR#
AO_RD#
WRFIFO#
RDFIFO#
HCS#
HDS1#
HRW
TBCA0/HBE0#
TBCA1/HBE1#
TBCA2/HHWIL
TBCA3/HCNTL0
TBCA4/HCNTL1
TBC_RD#
TBC_WR#
113
197
108
109
99
69
101
102
92
169
170
64
123
124
133
78
76
137
79
194
130
81
192
188
121
187
VCC3
VCC5
R513
10K
R520
10K
TBC_INT#
HINT#
TBC_RDY#
HRDY#
PCI_IRQ#
DB_INT
DSP_HINT#
201
205
9
40
4
96
EPM7256SQC208
TEXAS INSTRUMENTS INCORPORATED
Title
TMS320C6X EVM
Size
A
Document Number
D600832
Rev
*
Date: Tuesday, February 10, 1998
Sheet 25 of 40
PLD

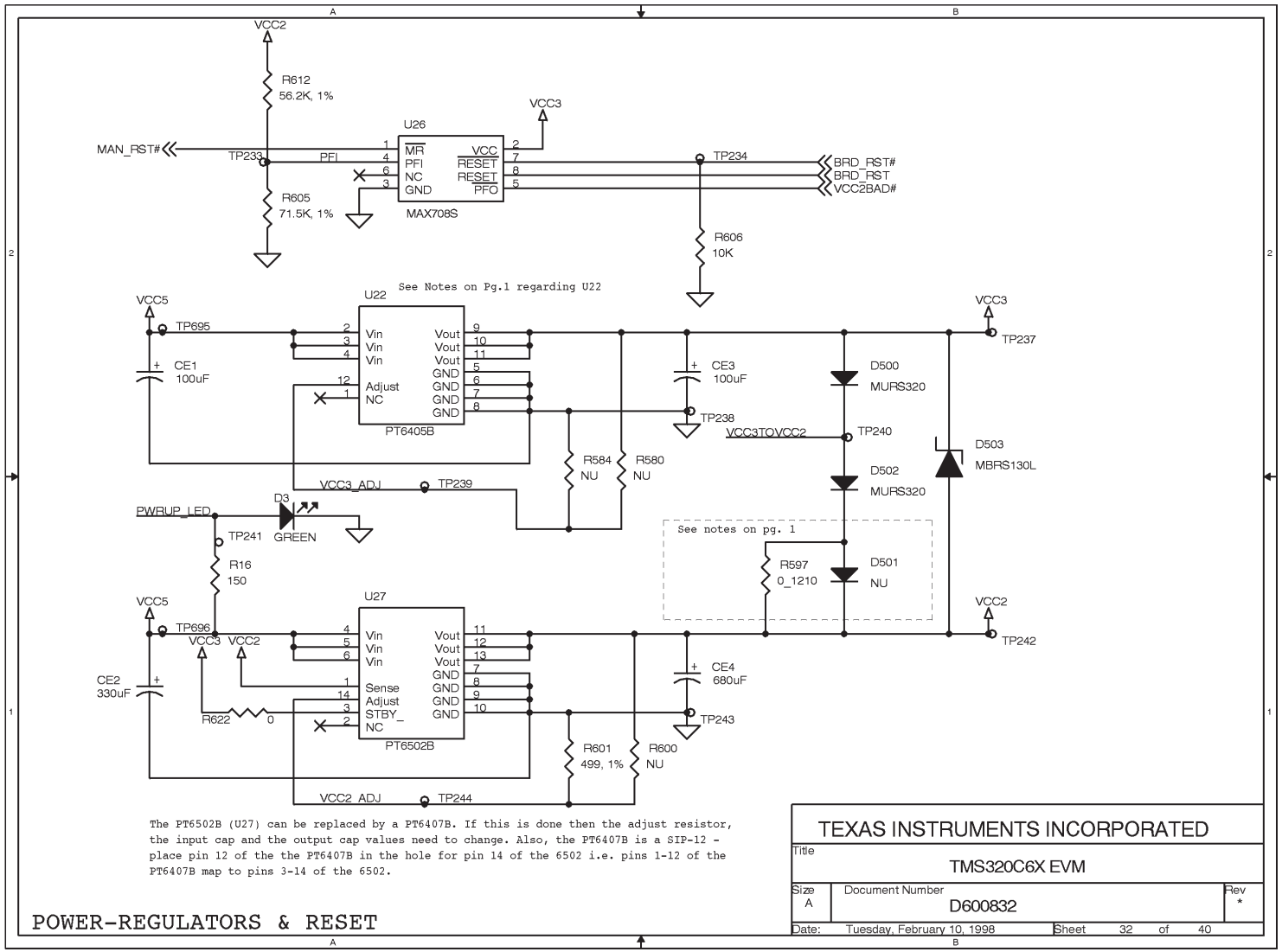

PBA[21..16]
PBA[6..2]
PBD[7..0]
U12C
PBA21 173 EA21
PBA20 178 EA20
PBA19 161 EA19
PBA18 162 EA18
PBA17 166 EA17
PBA16 34 EA16
PBA6 117 EA6
PBA5 140 EA5
PBA4 139 EA4
PBA3 97 EA3
PBA2 126 EA2
ED7 199 PBD7
ED6 8 PBD6
ED5 77 PBD5
ED4 206 PBD4
ED3 202 PBD3
ED2 11 PBD2
ED1 48 PBD1
ED0 62 PBD0
VCC3
R564
10K
CE1# 142 CE1#
ARDY 59 ARDY
DSP2XD# 60 DSP2XD#
DSP2PBD# 203 DSP2PBD#
DSP2AOD# 10 DSP2AOD#
AWE# 181 AWE#
ARE# 3 ARE#
XRDY 160 XRDY
AO_SEL# 84 PCI_CS#
PLDCLK 147 DSPCLK
CODEC_CS# 58 CODEC_CS#
CODEC_WR# 68 CODEC_WE#
CODEC_RD# 198 CODEC_RE#
PBBE3# 136 PBBE3#
PBBE2# 110 PBBE2#
PBBE1# 129 PBBE1#
PBBE0# 100 PBBE0#
CE2# 128 CE2#
CE3# 86 CE3#
CE2_SDEN 55 CE2_SDEN
CE3_SDEN 56 CE3_SDEN
CODEC_IRQ 119 CODEC_IRQ
PC_INT 122 PC_INT
NMI 204 NMI
PCIMWR_INT 61 PCIMWR_INT
PCIMRD_INT 42 PCIMRD_INT
PCI_INT 131 PCI_INT
XSTAT1 112 XSTAT1
XSTAT0 98 XSTAT0
XCNTL1 15 XCNTL1
XCNTL0 6 XCNTL0
EPM7256SQC208
PLD
TEXAS INSTRUMENTS INCORPORATED
Title
TMS320C6X EVM
Size A
Document Number D600832
Rev *
Date: Tuesday, February 10, 1998
Sheet 26 of 40

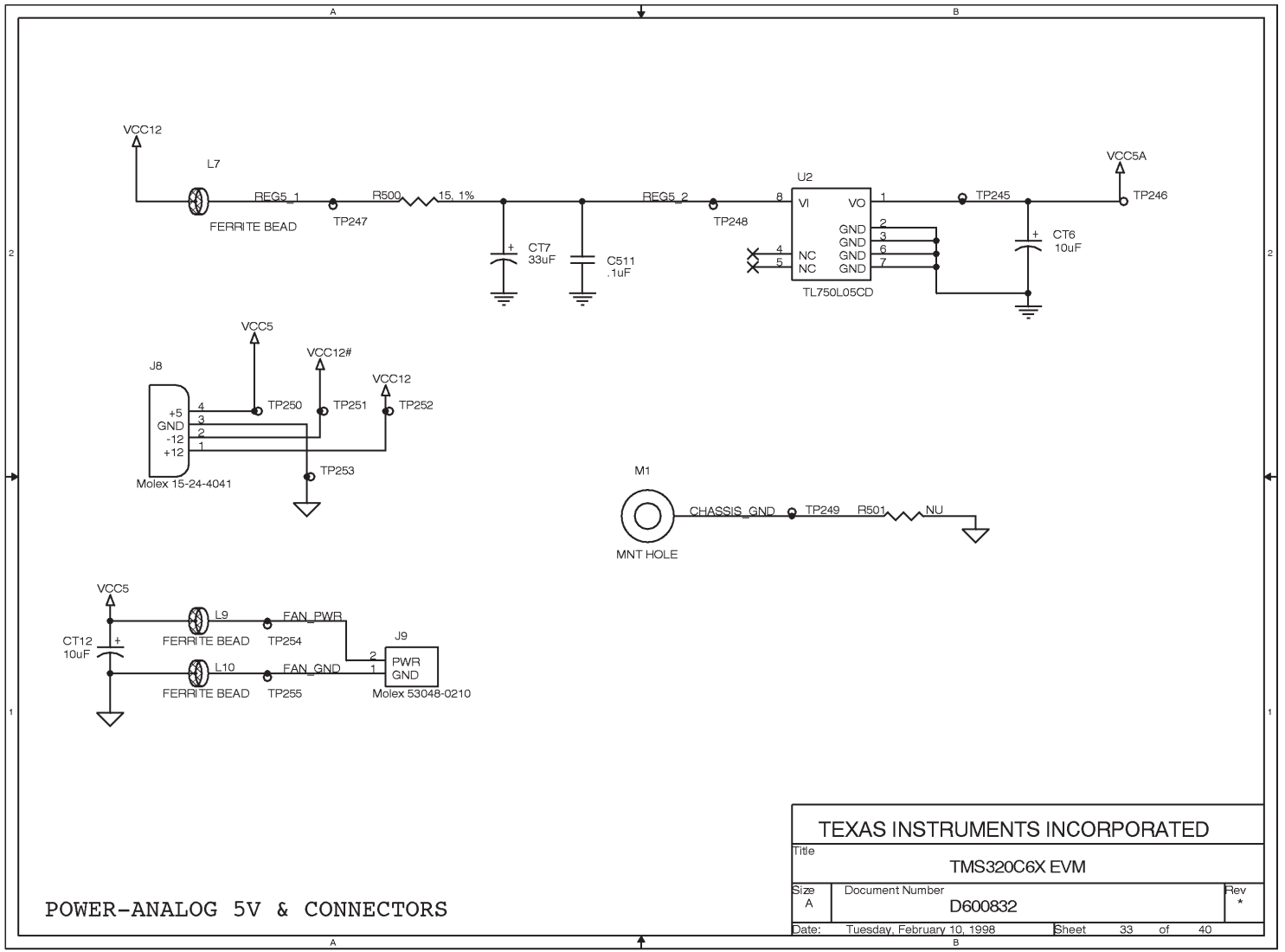

PCI_AD[31..0]
C/BE3#
C/BE2#
C/BE1#
C/BE0#
PAR
PCI_CLK
RST#
FRAME#
IRDY#
TRDY#
STOP#
LOCK#
IDSEL
DEVSEL#
REQ#
GNT#
PERR#
SERR#
INTA#
PCI_DET#
P1A
P1B
P1C
P1D
VCC3
VCC5
VCC12
VCC12#
R504
10K
TDI_TDO
INTA#
PCI_DET#
RST#
GNT#
PCI_AD30
TP216 A21
PCI_AD28
PCI_AD26
PCI_AD24
IDSEL
TP220 A27
PCI_AD22
PCI_AD20
PCI_AD18
PCI_AD16
TP207 A33
FRAME#
TRDY#
STOP#
TP210 A39
PAR
PCI_AD15
TP215 A45
PCI_AD13
PCI_AD11
PCI_AD9
C/BE0#
TP217 A53
PCI_AD6
PCI_AD4
PCI_AD2
PCI_AD0
TP208 TDI_TDO
TP211 PRSNT1#
TP213 PRSNT2#
PCI_CLK
REQ#
PCI_AD31
PCI_AD29
PCI_AD27
PCI_AD25
TP219 B25
C/BE3#
PCI_AD23
PCI_AD21
PCI_AD19
TP221 B31
PCI_AD17
C/BE2#
IRDY#
TP209 B36
DEVSEL#
LOCK#
PERR#
TP212 B41
SERR#
TP214 B43
C/BE1#
PCI_AD14
PCI_AD12
PCI_AD10
PCI_AD8
PCI_AD7
TP218 B54
PCI_AD5
PCI_AD3
PCI_AD1
PCI BUS CONNECTOR
PRSNT1# R502 0
PRSNT2# R503 NU
Indicates 25W Maximum
A21 C520 .01uF
A27 C524 .01uF
A33 C526 .01uF
A39 C530 .01uF
A45 C536 .01uF
A53 C539 .01uF
B25 C521 .01uF
B31 C525 .01uF
B36 C529 .01uF
B41 C532 .01uF
B43 C535 .01uF
B54 C541 .01uF
PCI CONNECTOR
TEXAS INSTRUMENTS INCORPORATED
Title
TMS320C6X EVM
Size A
Document Number D600832
Rev *
Date: Tuesday, February 10, 1998
Sheet 27 of 40

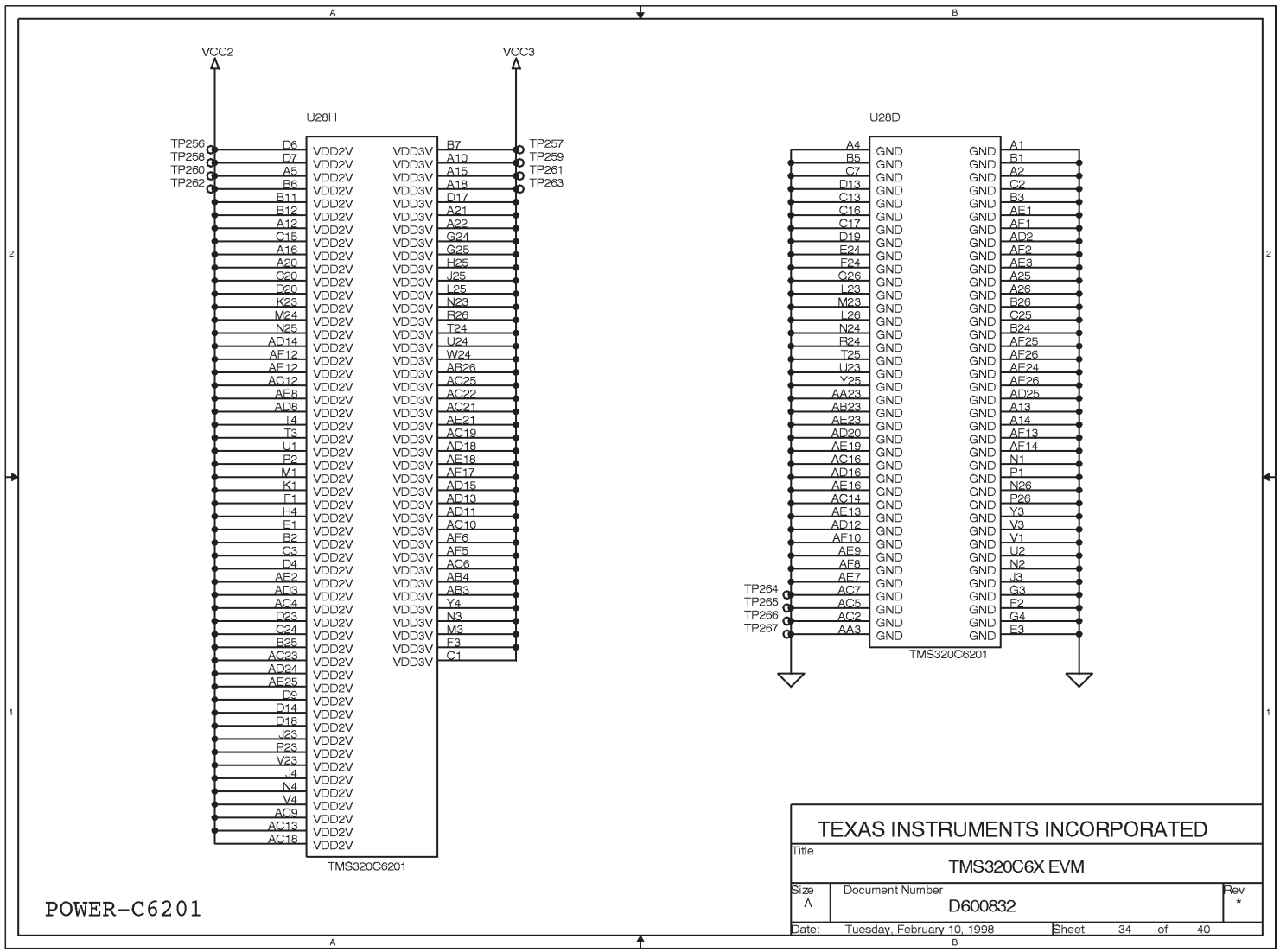

PCI_AD[31..0]
INTA#
SERR#
PERR#
GNT#
REQ#
DEVSEL#
IDSEL
LOCK#
STOP#
TRDY#
IRDY#
FRAME#
RST#
PCI_CLK
PAR
C/BE0#
C/BE1#
C/BE2#
C/BE3#
U5A
C/BE3# 159
C/BE2# 15
C/BE1# 28
C/BE0# 43
PAR 27
PCI_CLK 142
RST# 139
FRAME# 16
IRDY# 18
TRDY# 19
STOP# 22
LOCK# 23
IDSEL 160
DEVSEL# 20
REQ# 144
GNT# 143
PERR# 24
SERR# 26
INTA# 58
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
S5933
PCI-CONTROLLER BUS INTERFACE
TEXAS INSTRUMENTS INCORPORATED
Title
TMS320C6X EVM
Size A
Document Number
D600832
Rev *
Date: Tuesday, February 10, 1998
Sheet 28 of 40

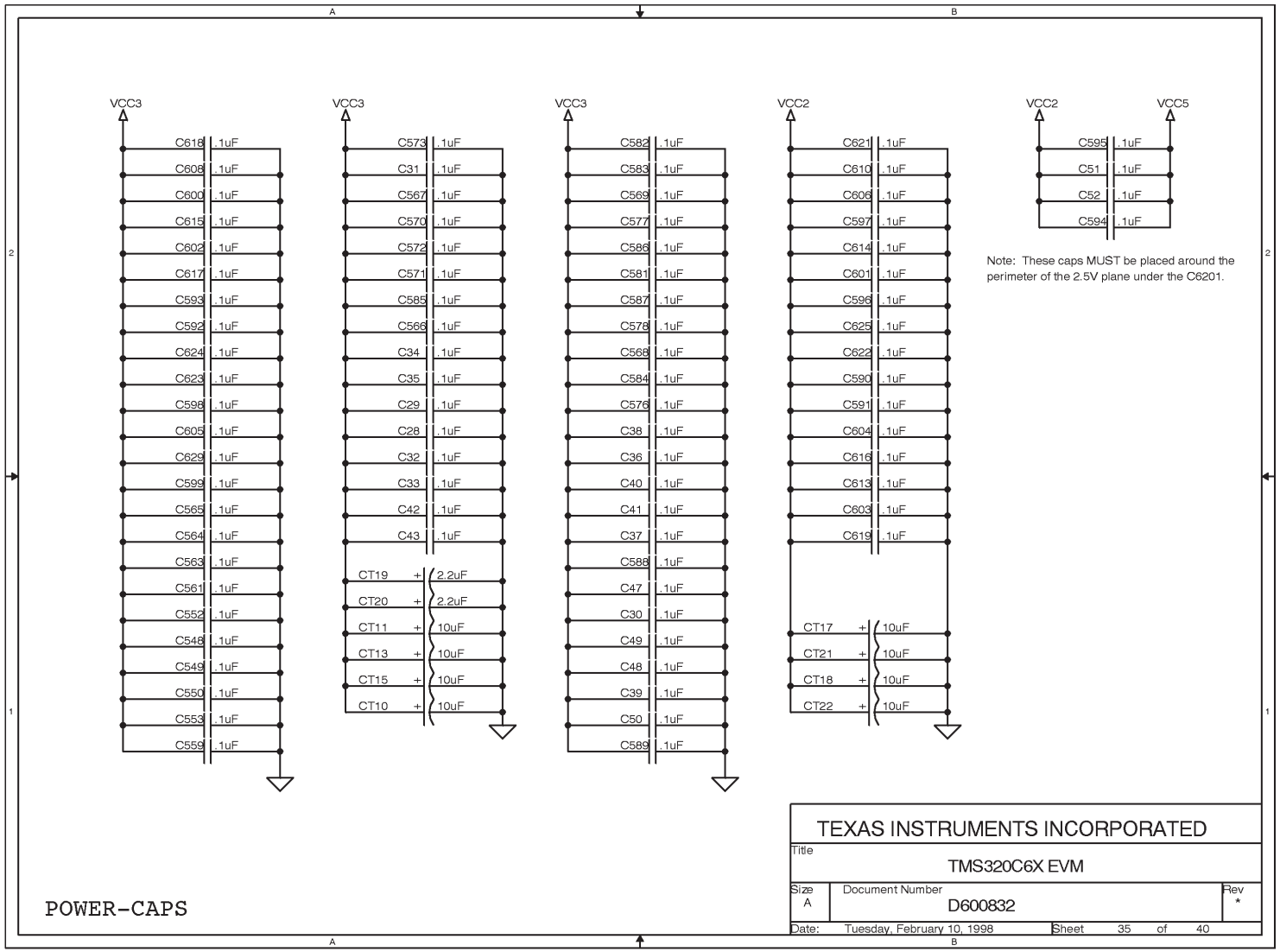

VCC5
U5B
AOD[31..0]
WRFIFO#
RDFIFO#
AMWEN
AMREN
FRC#
FWC#
WRFULL
RDEMPTY
FRF
FWE
AOA[6..2]
ADR6
ADR5
ADR4
ADR3
ADR2
BE3#
BE2#
BE1#
BE0#
SELECT#
WR#
RD#
MODE 16/32
PTRDY#
PTADR#
HINT#
TBC_INT#
EMB0
EMB1
EMB2
EMB3
EMB4
EMB5
EMB6
EMB7
EMBCLK
PC_INT
PCI_FLT#
FLT#
SNV
SCL
SDA
PTATN#
PTBURST#
PTNUM1
PTNUM0
PTBE3#
PTBE2#
PTBE1#
PTBE0#
PTWR
SYSRST#
BPCLK
IRQ#
PCI_SYSRST#
PCI_IRQ#
S5933
RN6
RN5
RN2
RN1
CTS 745 BUSSED 10K
R515
R511
R512
R510
R509
R507
10K
TP222
TP223
TP224
TP225
TP226
TP227
R19
R20
R505
TP228
U4
AT24C08A
TP230
TP229
TP231
R506
120
TP232
R508
120
PCI-CONTROLLER ADD-ON INTERFACE
TEXAS INSTRUMENTS INCORPORATED
TMS320C6X EVM
D600832
Tuesday, February 10, 1998
Sheet 29 of 40

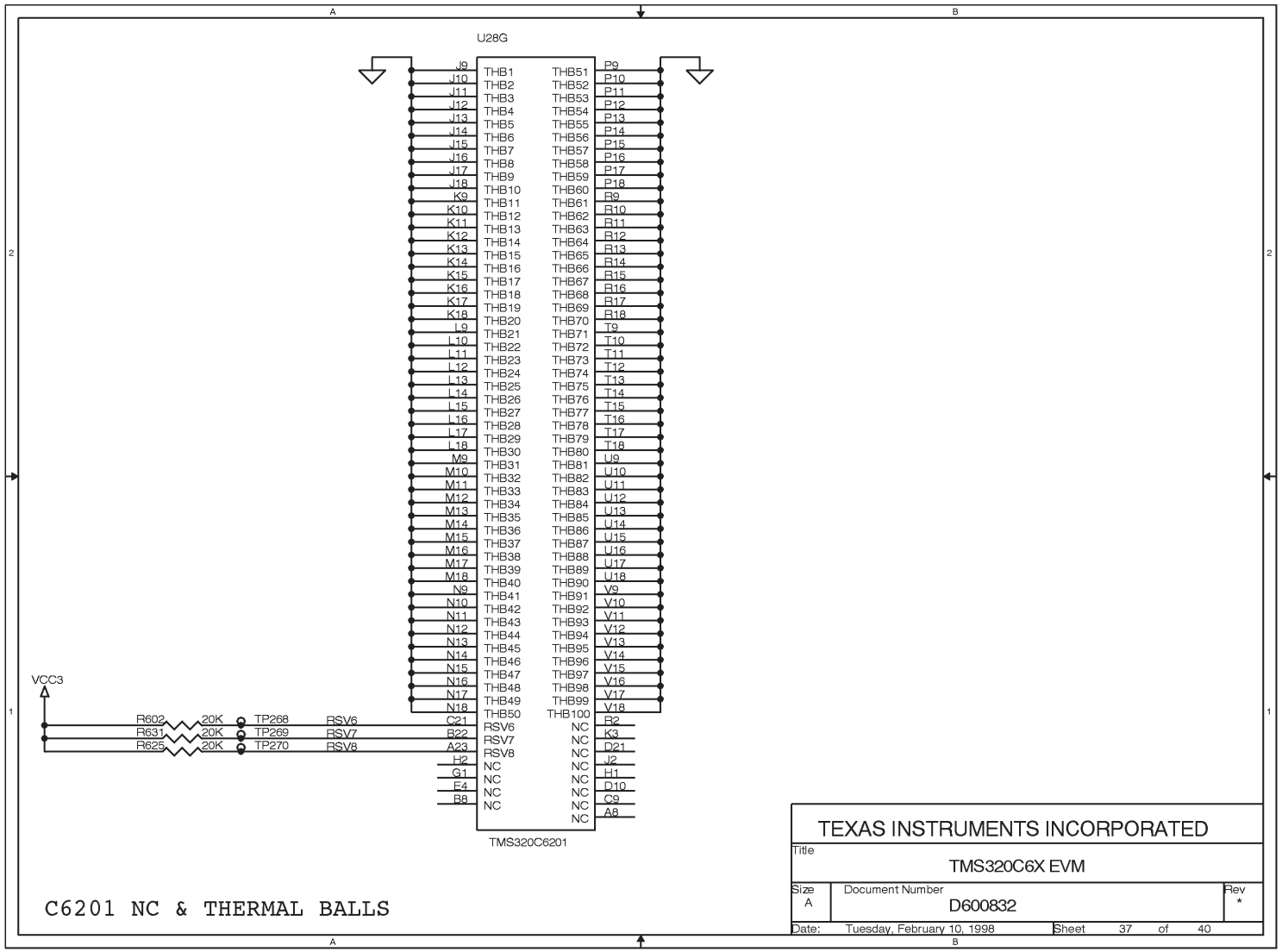

A
B
2
1
J6A
VCC5
VCC3
SFM-140-L2-S-D-LC
XA21
XA19
XA17
XA15
XA13
XA11
XA9
XA7
XA5
XA3
XBE3#
XBE1#
XD31
XD29
XD27
XD25
XD23
XD21
XD19
XD17
XD15
XD13
XD11
XD9
XD7
XD5
XD3
XD1
XRE#
XOE#
J6B
XA[21..2]
XD[31..0]
XA20
XA18
XA16
XA14
XA12
XA10
XA8
XA6
XA4
XA2
XBE2#
XBE0#
XD30
XD28
XD26
XD24
XD22
XD20
XD18
XD16
XD14
XD12
XD10
XD8
XD6
XD4
XD2
XD0
XWE#
XRDY
XCE1#
DB-EXPANSION MEMORY INTERFACE
TEXAS INSTRUMENTS INCORPORATED
Title
TMS320C6X EVM
Size
A
Document Number
D600832
Rev
*
Date: Tuesday, February 10, 1998
Sheet 30 of 40

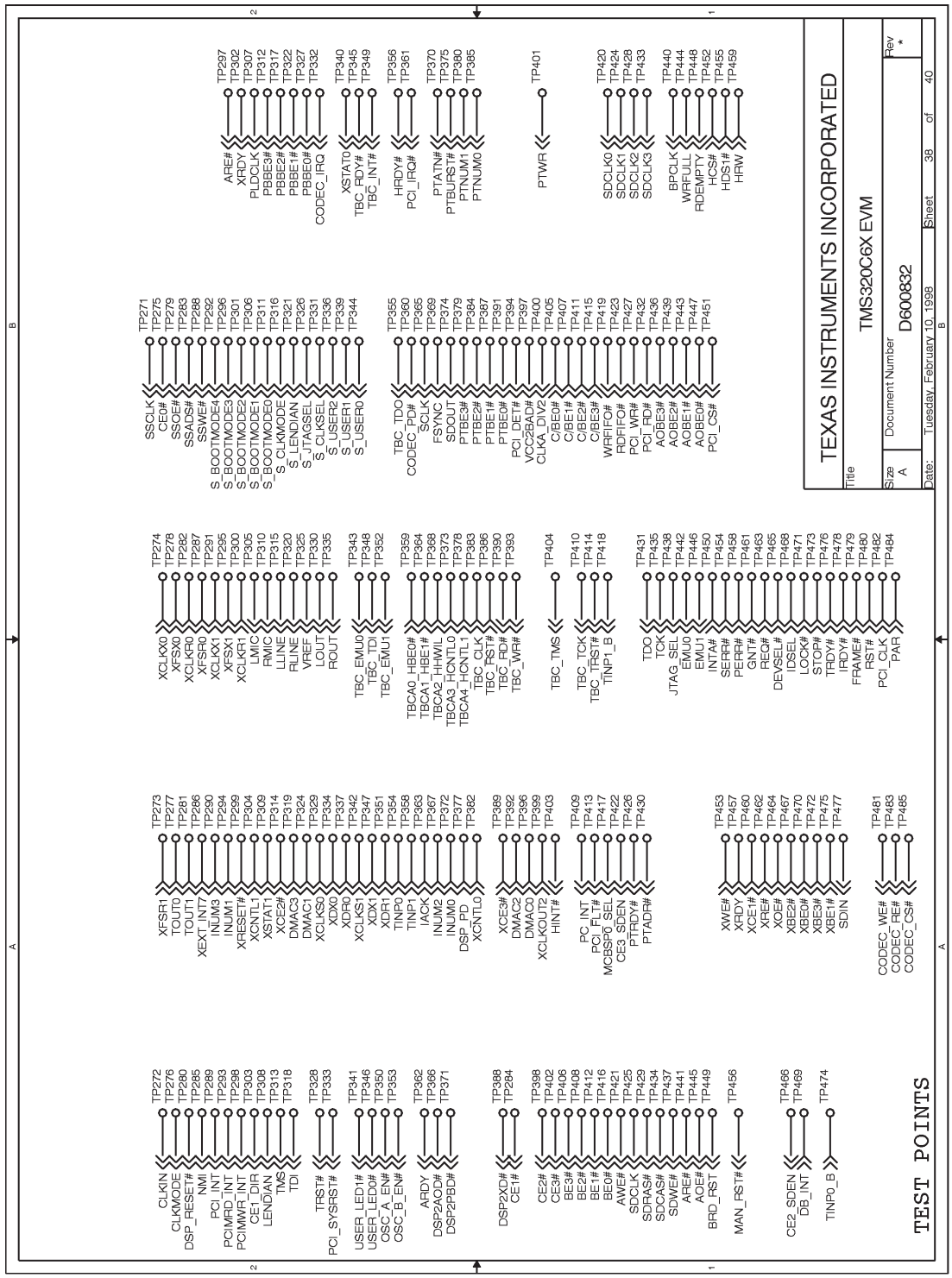

VCC2
R612
56.2K, 1%
VCC3
U26
MAN_RST#
TP233
PFI
MR
PFI
NC
GND
VCC
RESET
RESET
PFO
MAX708S
TP234
BRD_RST#
BRD_RST
VCC2BAD#
R605
71.5K, 1%
R606
10K
See Notes on Pg.1 regarding U22
VCC5
TP695
U22
Vin
Vin
Vin
Adjust
NC
Vout
Vout
Vout
GND
GND
GND
GND
PT6405B
CE1
100uF
CE3
100uF
VCC3
TP237
D500
MURS320
TP238
VCC3TOVCC2
TP240
D503
MBRS130L
D502
MURS320
R584
NU
R580
NU
VCC3_ADJ
TP239
PWRUP_LED
D3
GREEN
TP241
R16
150
See notes on pg. 1
R597
0_1210
D501
NU
VCC5
TP696
U27
VCC3
VCC2
Sense
Adjust
STBY_
NC
PT6502B
CE2
330uF
R622
0
CE4
680uF
TP243
VCC2
TP242
R601
499, 1%
R600
NU
VCC2_ADJ
TP244
The PT6502B (U27) can be replaced by a PT6407B. If this is done then the adjust resistor, the input cap and the output cap values need to change. Also, the PT6407B is a SIP-12 - place pin 12 of the the PT6407B in the hole for pin 14 of the 6502 i.e. pins 1-12 of the PT6407B map to pins 3-14 of the 6502.
TEXAS INSTRUMENTS INCORPORATED
Title
TMS320C6X EVM
Size
A
Document Number
D600832
Rev
*
Date:
Tuesday, February 10, 1998
Sheet
32
of
40
POWER-REGULATORS & RESET

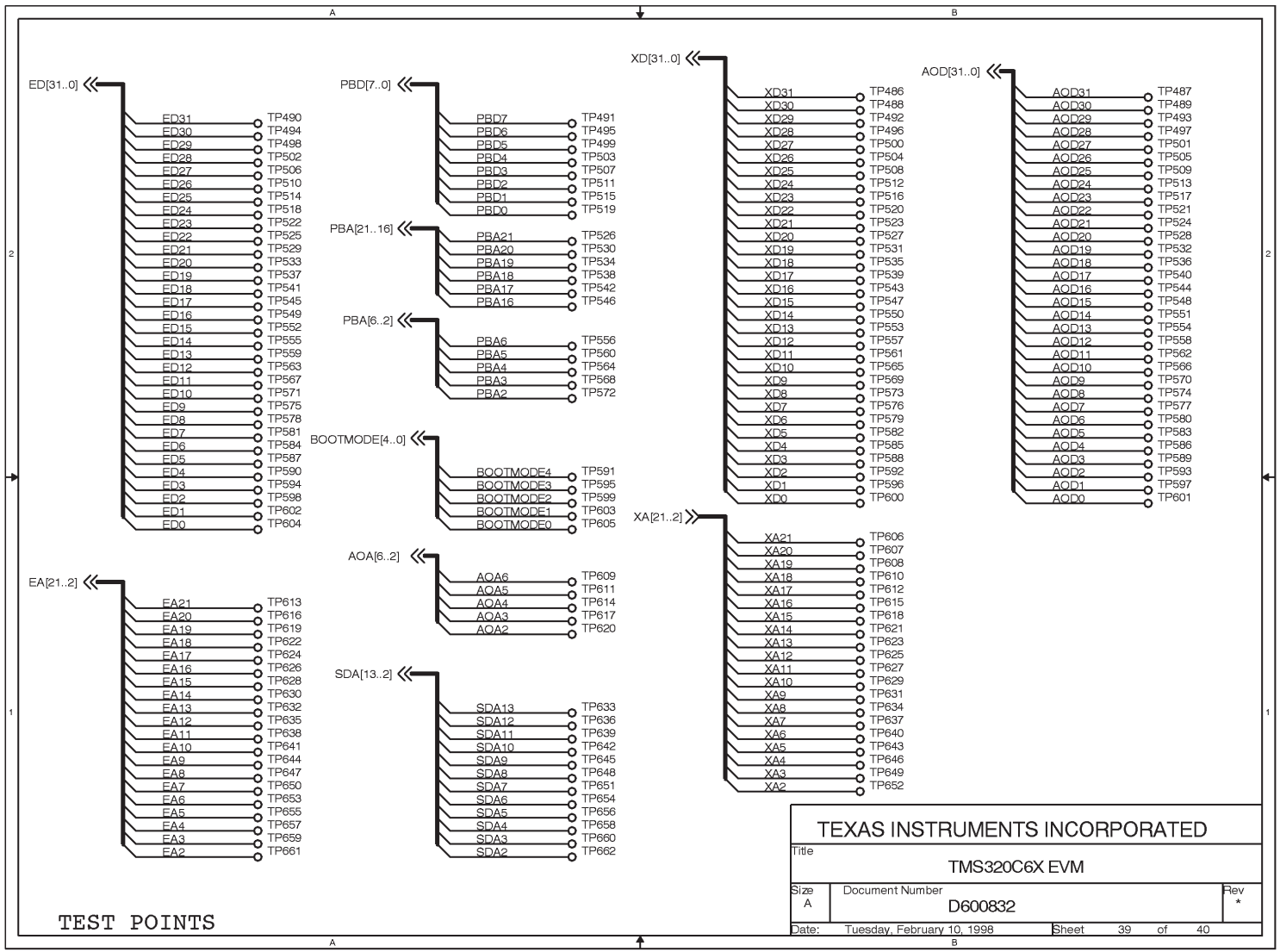

VCC12
L7
REG5_1
FERRITE BEAD
TP247
R500
15, 1%
REG5_2
TP248
CT7
33uF
C511
.1uF
U2
VI
VO
GND
NC
TL750L05CD
TP245
VCC5A
TP246
CT6
10uF
VCC5
VCC12#
VCC12
J8
+5
GND
-12
+12
TP250
TP251
TP252
TP253
Molex 15-24-4041
M1
MNT HOLE
CHASSIS_GND
TP249
R501
NU
CT12
10uF
L9
L10
FERRITE BEAD
FAN_PWR
FAN_GND
TP254
TP255
J9
PWR
GND
Molex 53048-0210
POWER-ANALOG 5V & CONNECTORS
TEXAS INSTRUMENTS INCORPORATED
Title
TMS320C6X EVM
Size
A
Document Number
D600832
Rev
*
Date: Tuesday, February 10, 1998
Sheet 33 of 40

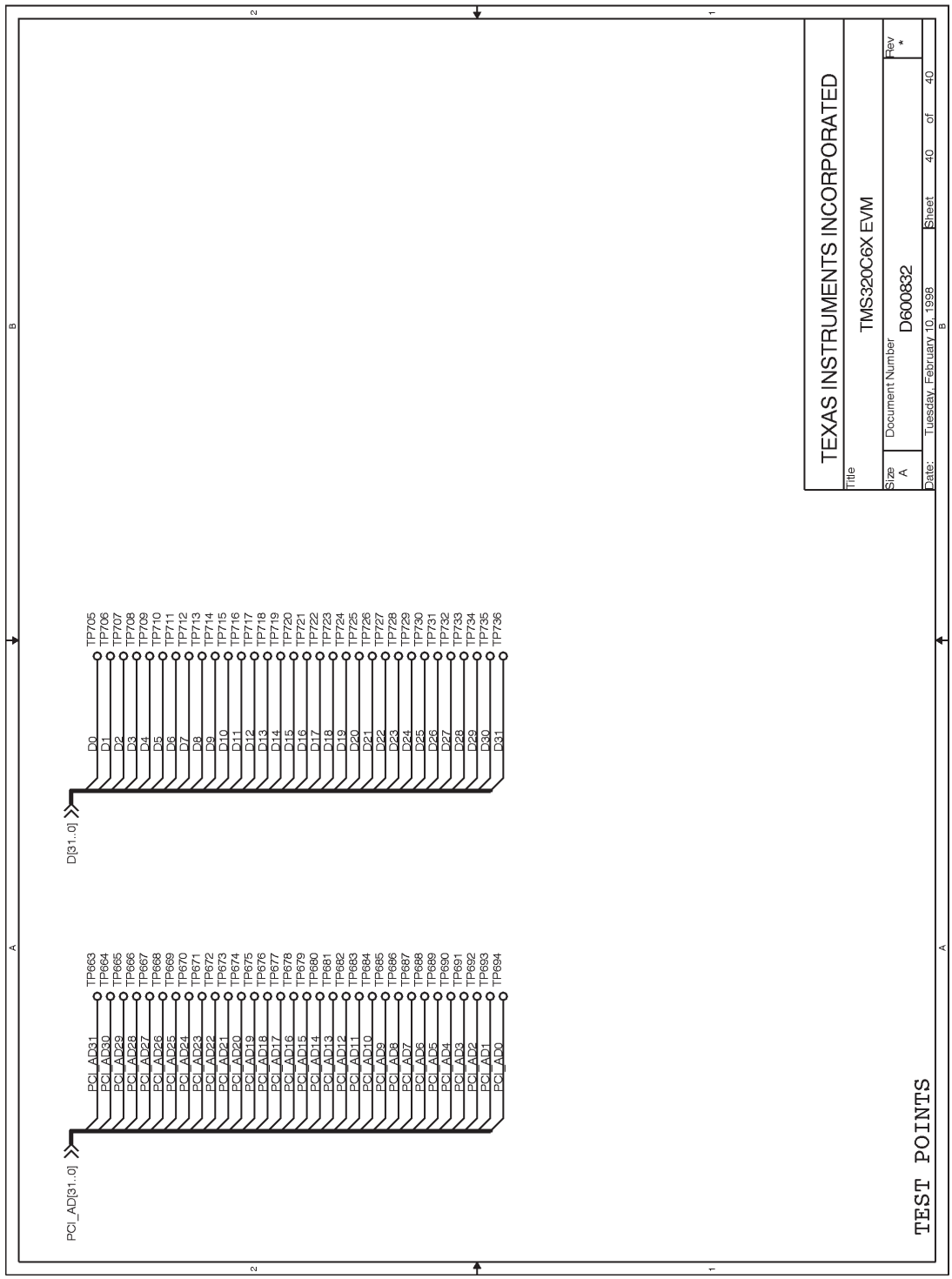

VCC2
VCC3
U28H
TP256
TP258
TP260
TP262
TP257
TP259
TP261
TP263
TMS320C6201
U28D
TP264
TP265
TP266
TP267
TMS320C6201
POWER-C6201
TEXAS INSTRUMENTS INCORPORATED
Title
TMS320C6X EVM
Size
A
Document Number
D600832
Rev
*
Date: Tuesday, February 10, 1998
Sheet 34 of 40


VCC3
C616 .1uF
C608 .1uF
C600 .1uF
C615 .1uF
C602 .1uF
C617 .1uF
C593 .1uF
C592 .1uF
C624 .1uF
C623 .1uF
C598 .1uF
C605 .1uF
C629 .1uF
C599 .1uF
C565 .1uF
C564 .1uF
C563 .1uF
C561 .1uF
C552 .1uF
C548 .1uF
C549 .1uF
C550 .1uF
C553 .1uF
C559 .1uF
VCC3
C573 .1uF
C31 .1uF
C567 .1uF
C570 .1uF
C572 .1uF
C571 .1uF
C585 .1uF
C566 .1uF
C34 .1uF
C35 .1uF
C29 .1uF
C28 .1uF
C32 .1uF
C33 .1uF
C42 .1uF
C43 .1uF
CT19 + 2.2uF
CT20 + 2.2uF
CT11 + 10uF
CT13 + 10uF
CT15 + 10uF
CT10 + 10uF
VCC3
C582 .1uF
C583 .1uF
C569 .1uF
C577 .1uF
C586 .1uF
C581 .1uF
C587 .1uF
C578 .1uF
C568 .1uF
C584 .1uF
C576 .1uF
C38 .1uF
C36 .1uF
C40 .1uF
C41 .1uF
C37 .1uF
C588 .1uF
C47 .1uF
C30 .1uF
C49 .1uF
C48 .1uF
C39 .1uF
C50 .1uF
C589 .1uF
VCC2
C621 .1uF
C610 .1uF
C606 .1uF
C597 .1uF
C614 .1uF
C601 .1uF
C596 .1uF
C625 .1uF
C622 .1uF
C590 .1uF
C591 .1uF
C604 .1uF
C616 .1uF
C613 .1uF
C603 .1uF
C619 .1uF
CT17 + 10uF
CT21 + 10uF
CT18 + 10uF
CT22 + 10uF
VCC2
VCC5
C595 .1uF
C51 .1uF
C52 .1uF
C594 .1uF
Note: These caps MUST be placed around the perimeter of the 2.5V plane under the C6201.
TEXAS INSTRUMENTS INCORPORATED
Title
TMS320C6X EVM
Size
A
Document Number
D600832
Rev
*
Date: Tuesday, February 10, 1998
Sheet 35 of 40
POWER-CAPS


U28G
J9 THB1 THB51 P9
J10 THB2 THB52 P10
J11 THB3 THB53 P11
J12 THB4 THB54 P12
J13 THB5 THB55 P13
J14 THB6 THB56 P14
J15 THB7 THB57 P15
J16 THB8 THB58 P16
J17 THB9 THB59 P17
J18 THB10 THB60 P18
K9 THB11 THB61 R9
K10 THB12 THB62 R10
K11 THB13 THB63 R11
K12 THB14 THB64 R12
K13 THB15 THB65 R13
K14 THB16 THB66 R14
K15 THB17 THB67 R15
K16 THB18 THB68 R16
K17 THB19 THB69 R17
K18 THB20 THB70 R18
L9 THB21 THB71 T9
L10 THB22 THB72 T10
L11 THB23 THB73 T11
L12 THB24 THB74 T12
L13 THB25 THB75 T13
L14 THB26 THB76 T14
L15 THB27 THB77 T15
L16 THB28 THB78 T16
L17 THB29 THB79 T17
L18 THB30 THB80 T18
M9 THB31 THB81 U9
M10 THB32 THB82 U10
M11 THB33 THB83 U11
M12 THB34 THB84 U12
M13 THB35 THB85 U13
M14 THB36 THB86 U14
M15 THB37 THB87 U15
M16 THB38 THB88 U16
M17 THB39 THB89 U17
M18 THB40 THB90 U18
N9 THB41 THB91 V9
N10 THB42 THB92 V10
N11 THB43 THB93 V11
N12 THB44 THB94 V12
N13 THB45 THB95 V13
N14 THB46 THB96 V14
N15 THB47 THB97 V15
N16 THB48 THB98 V16
N17 THB49 THB99 V17
N18 THB50 THB100 V18
RSV6 NC R2
RSV7 NC K3
RSV8 NC D21
NC NC J2
NC NC H1
NC NC D10
NC NC C9
NC NC A8
TMS320C6201
VCC3
R602 20K TP268 RSV6 C21
R631 20K TP269 RSV7 B22
R625 20K TP270 RSV8 A23
H2
G1
E4
B8
C6201 NC & THERMAL BALLS
TEXAS INSTRUMENTS INCORPORATED
Title
TMS320C6X EVM
Size A
Document Number D600832
Rev *
Date: Tuesday, February 10, 1998
Sheet 37 of 40


TEST POINTS
TEXAS INSTRUMENTS INCORPORATED
TMS320C6X EVM
Document Number
D600832
Tuesday, February 10, 1998
Sheet 38 of 40
Rev *


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
PBD[7..0]
PBD7 TP491
PBD6 TP495
PBD5 TP499
PBD4 TP503
PBD3 TP507
PBD2 TP511
PBD1 TP515
PBD0 TP519
PBA[21..16]
PBA21 TP526
PBA20 TP530
PBA19 TP534
PBA18 TP538
PBA17 TP542
PBA16 TP546
PBA[6..2]
PBA6 TP556
PBA5 TP560
PBA4 TP564
PBA3 TP568
PBA2 TP572
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[13..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
TEXAS INSTRUMENTS INCORPORATED
Title
TMS320C6X EVM
Size A
Document Number
D600832
Rev *
Date: Tuesday, February 10, 1998
Sheet 39 of 40


TEXAS INSTRUMENTS INCORPORATED
TMS320C6X EVM
Document Number
D600832
Rev
*
Size
A
Date: Tuesday, February 10, 1998
Sheet 40 of 40
TEST POINTS
PCI_AD[31..0]
PCI_AD31 TP663
PCI_AD30 TP664
PCI_AD29 TP665
PCI_AD28 TP666
PCI_AD27 TP667
PCI_AD26 TP668
PCI_AD25 TP669
PCI_AD24 TP670
PCI_AD23 TP671
PCI_AD22 TP672
PCI_AD21 TP673
PCI_AD20 TP674
PCI_AD19 TP675
PCI_AD18 TP676
PCI_AD17 TP677
PCI_AD16 TP678
PCI_AD15 TP679
PCI_AD14 TP680
PCI_AD13 TP681
PCI_AD12 TP682
PCI_AD11 TP683
PCI_AD10 TP684
PCI_AD9 TP685
PCI_AD8 TP686
PCI_AD7 TP687
PCI_AD6 TP688
PCI_AD5 TP689
PCI_AD4 TP690
PCI_AD3 TP691
PCI_AD2 TP692
PCI_AD1 TP693
PCI_AD0 TP694
D[31..0]
D0 TP705
D1 TP706
D2 TP707
D3 TP708
D4 TP709
D5 TP710
D6 TP711
D7 TP712
D8 TP713
D9 TP714
D10 TP715
D11 TP716
D12 TP717
D13 TP718
D14 TP719
D15 TP720
D16 TP721
D17 TP722
D18 TP723
D19 TP724
D20 TP725
D21 TP726
D22 TP727
D23 TP728
D24 TP729
D25 TP730
D26 TP731
D27 TP732
D28 TP733
D29 TP734
D30 TP735
D31 TP736

Appendix C

TMS320C6x EVM CPLD Equations

This appendix provides the complex programmable logic device (CPLD) equations. The CPLD, which is the only programmable logic device on the ’C6x EVM, is designated as U12 on the board.

Topic **Page**

C.1 Overview of the EVM 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. TMS320C6x EVM 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 ABEL 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, *EVM CPLD Equations*.

Figure C–1. TMS320C6x EVM CPLD Source Files

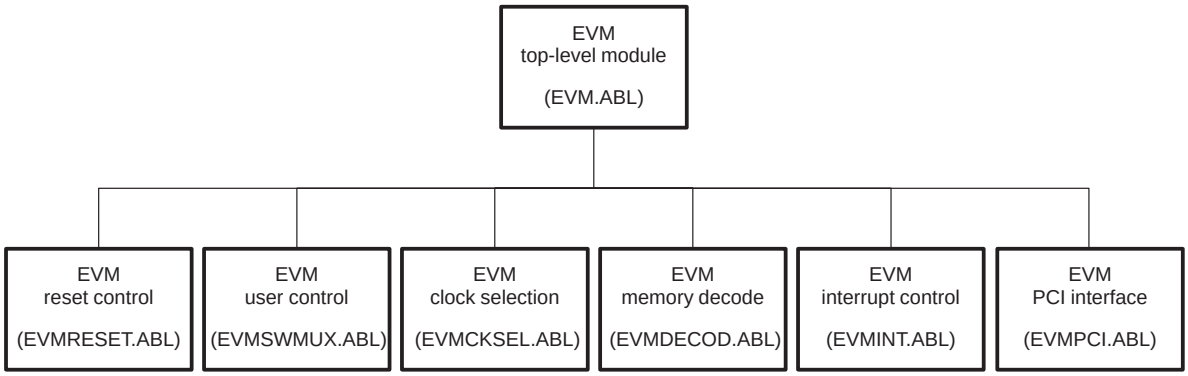


Table C–2 provides the CPLD pin definitions in numerical pin order. Table C–3 provides the CPLD pin definitions in sequential alphabetical order.

Table C–2. TMS320C6x CPLD Pin Definitions (Numerical Pin Order)

Pin	Signal	Pin	Signal	Pin	Signal	Pin	Signal
1	N/C	27	BOOTMODE4	53	N/C	79	HRW
2	N/C	28	Reserved	54	N/C	80	PTNUM0
3	$\overline{\text{ARE}}$	29	DQ6	55	CE2_SDEN	81	TBCA2_HHWIL
4	$\overline{\text{PCI\_IRQ}}$	30	TCK	56	CE3_SDEN	82	GND
5	VCC3	31	Reserved	57	Reserved	83	VCC5
6	XCNTL0	32	GND	58	$\overline{\text{CODEC\_CS}}$	84	$\overline{\text{AO\_SEL}}$
7	Reserved	33	Reserved	59	ARDY	85	VCC3
8	ED6	34	EA16	60	$\overline{\text{DSP2XD}}$	86	$\overline{\text{CE3}}$
9	$\overline{\text{DSP\_HINT}}$	35	BOOTMODE3	61	PCIMWR_INT	87	Reserved
10	$\overline{\text{DSP2AOD}}$	36	DQ3	62	ED0	88	Reserved
11	ED2	37	CLKMODE	63	VCC3	89	S_BOOTMODE0
12	SP0SEL	38	JTAGSEL	64	$\overline{\text{PCI\_FLT}}$	90	RDEMPTY
13	BOOTMODE2	39	Reserved	65	$\overline{\text{LED0}}$	91	Reserved
14	GND	40	$\overline{\text{DSP\_HRDY}}$	66	Reserved	92	AO_ADR4
15	XCNTL1	41	VCC3	67	Reserved	93	S_USER1
16	DQ2	42	PCIMRD_INT	68	$\overline{\text{CODEC\_WR}}$	94	GND
17	Reserved	43	DQ7	69	$\overline{\text{AO\_BE0}}$	95	S_USER2
18	Reserved	44	$\overline{\text{TBC\_RST}}$	70	$\overline{\text{EXT\_RST}}$	96	DB_INT
19	DQ4	45	Reserved	71	$\overline{\text{OSC\_A\_EN}}$	97	EA3
20	Reserved	46	BOOTMODE1	72	GND	98	XSTAT0
21	Reserved	47	DQ1	73	$\overline{\text{OSC\_B\_EN}}$	99	$\overline{\text{AO\_BE1}}$
22	DQ0	48	ED1	74	VCC5	100	$\overline{\text{BE0}}$
23	VCC3	49	$\overline{\text{LED1}}$	75	GND	101	AO_ADR6
24	BOOTMODE0	50	GND	76	$\overline{\text{HCS}}$	102	AO_ADR5
25	$\overline{\text{PTBE0}}$	51	N/C	77	ED5	103	N/C
26	LENDIAN	52	N/C	78	$\overline{\text{RDFIFO}}$	104	N/C

Table C–2. TMS320C6x CPLD Pin Definitions (Numerical Pin Order) (Continued)

Pin	Signal	Pin	Signal	Pin	Signal	Pin	Signal
105	N/C	131	PCI_INT	157	N/C	183	$\overline{\text{FLOAT}}$
106	N/C	132	$\overline{\text{MAN\_RST}}$	158	N/C	184	BPCLK
107	VCC3	133	$\overline{\text{WRFIFO}}$	159	S_CLKSEL	185	GND
108	$\overline{\text{AO\_BE3}}$	134	GND	160	XRDY	186	VCC5
109	AO_BE2	135	$\overline{\text{DSP\_RST}}$	161	EA19	187	$\overline{\text{TBC\_WR}}$
110	$\overline{\text{BE2}}$	136	$\overline{\text{BE3}}$	162	EA18	188	TBCA4_HCNTL1
111	PTWR	137	$\overline{\text{HDS1}}$	163	Reserved	189	TDO
112	XSTAT1	138	$\overline{\text{CODEC\_PD}}$	164	S_BOOTMODE4	190	$\overline{\text{PTBE3}}$
113	$\overline{\text{PT\_RDY}}$	139	EA4	165	VCC3	191	VCC3
114	Reserved	140	EA5	166	EA17	192	TBCA3_HCNTL0
115	CLKA_DIV2	141	S_ENDIAN	167	$\overline{\text{PCI\_DET}}$	193	$\overline{\text{PTBE2}}$
116	GND	142	$\overline{\text{CE1}}$	168	PTNUM1	194	$\overline{\text{TBCA0\_HBE0}}$
117	EA6	143	VCC3	169	AO_ADR3	195	$\overline{\text{PTBE1}}$
118	S_BOOTMODE2	144	DQ5	170	AO_ADR2	196	Reserved
119	CODEC_IRQ	145	S_JTAGSEL	171	Reserved	197	$\overline{\text{PT\_ADR}}$
120	$\overline{\text{SW\_RST}}$	146	S_CLKMODE	172	WRFULL	198	$\overline{\text{CODEC\_RD}}$
121	$\overline{\text{TBC\_RD}}$	147	DSPCLK	173	EA21	199	ED7
122	PC_INT	148	S_BOOTMODE1	174	GND	200	GND
123	$\overline{\text{AO\_WR}}$	149	$\overline{\text{PTBURST}}$	175	$\overline{\text{VCC2BAD}}$	201	$\overline{\text{TBC\_INT}}$
124	$\overline{\text{AO\_RD}}$	150	S_USER0	176	TDI	202	ED3
125	VCC3	151	S_BOOTMODE3	177	DSP_PD	203	$\overline{\text{DSP2PBD}}$
126	EA2	152	GND	178	EA20	204	NMI
127	TMS	153	$\overline{\text{PCI\_RST}}$	179	VCC5	205	$\overline{\text{TBC\_RDY}}$
128	$\overline{\text{CE2}}$	154	$\overline{\text{PTATN}}$	180	GND	206	ED4
129	$\overline{\text{BE1}}$	155	N/C	181	$\overline{\text{AWE}}$	207	N/C
130	$\overline{\text{TBCA1\_HBE1}}$	156	N/C	182	$\overline{\text{BRD\_RST}}$	208	N/C

Table C–3. TMS320C6x CPLD Pin Definitions (Alphabetical Pin Order)

Signal	Pin	Signal	Pin	Signal	Pin	Signal	Pin
AO_ADR2	170	$\overline{\text{CE1}}$	142	$\overline{\text{DSP2PBD}}$	203	GND	50
AO_ADR3	169	$\overline{\text{CE2}}$	128	$\overline{\text{DSP2XD}}$	60	GND	72
AO_ADR4	92	CE2_SDEN	55	DSPCLK	147	GND	75
AO_ADR5	102	$\overline{\text{CE3}}$	86	EA16	34	GND	82
AO_ADR6	101	CE3_SDEN	56	EA17	166	GND	94
$\overline{\text{AO\_BE0}}$	69	CLKA_DIV2	115	EA18	162	GND	116
$\overline{\text{AO\_BE1}}$	99	CLKMODE	37	EA19	161	GND	134
AO_BE2	109	$\overline{\text{CODEC\_CS}}$	58	EA2	126	GND	152
$\overline{\text{AO\_BE3}}$	108	CODEC_IRQ	119	EA20	178	GND	174
$\overline{\text{AO\_RD}}$	124	$\overline{\text{CODEC\_PD}}$	138	EA21	173	GND	180
$\overline{\text{AO\_SEL}}$	84	$\overline{\text{CODEC\_RD}}$	198	EA3	97	GND	185
$\overline{\text{AO\_WR}}$	123	$\overline{\text{CODEC\_WR}}$	68	EA4	139	GND	200
ARDY	59	DB_INT	96	EA5	140	$\overline{\text{HCS}}$	76
$\overline{\text{ARE}}$	3	DQ0	22	EA6	117	$\overline{\text{HDS1}}$	137
$\overline{\text{AWE}}$	181	DQ1	47	ED0	62	HRW	79
$\overline{\text{BE0}}$	100	DQ2	16	ED1	48	JTAGSEL	38
$\overline{\text{BE1}}$	129	DQ3	36	ED2	11	$\overline{\text{LED0}}$	65
$\overline{\text{BE2}}$	110	DQ4	19	ED3	202	$\overline{\text{LED1}}$	49
$\overline{\text{BE3}}$	136	DQ5	144	ED4	206	LENDIAN	26
BOOTMODE0	24	DQ6	29	ED5	77	$\overline{\text{MAN\_RST}}$	132
BOOTMODE1	46	DQ7	43	ED6	8	N/C	1
BOOTMODE2	13	$\overline{\text{DSP\_HINT}}$	9	ED7	199	N/C	2
BOOTMODE3	35	$\overline{\text{DSP\_HRDY}}$	40	$\overline{\text{EXT\_RST}}$	70	N/C	51
BOOTMODE4	27	DSP_PD	177	$\overline{\text{FLOAT}}$	183	N/C	52
BPCLK	184	$\overline{\text{DSP\_RST}}$	135	GND	14	N/C	53
$\overline{\text{BRD\_RST}}$	182	$\overline{\text{DSP2AOD}}$	10	GND	32	N/C	54

Table C–3. TMS320C6x CPLD Pin Definitions (Alphabetical Pin Order) (Continued)

Signal	Pin	Signal	Pin	Signal	Pin	Signal	Pin
N/C	103	PTBE2	193	Reserved	171	TCK	30
N/C	104	PTBE3	190	Reserved	196	TDI	176
N/C	105	PTBURST	149	S_BOOTMODE0	89	TDO	189
N/C	106	PTNUM0	80	S_BOOTMODE1	148	TMS	127
N/C	155	PTNUM1	168	S_BOOTMODE2	118	VCC2BAD	175
N/C	156	PTWR	111	S_BOOTMODE3	151	VCC3	5
N/C	157	RDEMPTY	90	S_BOOTMODE4	164	VCC3	23
N/C	158	RDFIFO	78	S_CLKMODE	146	VCC3	41
N/C	207	Reserved	7	S_CLKSEL	159	VCC3	63
N/C	208	Reserved	17	S_ENDIAN	141	VCC3	85
NMI	204	Reserved	18	S_JTAGSEL	145	VCC3	107
OSC_A_EN	71	Reserved	20	S_USER0	150	VCC3	125
OSC_B_EN	73	Reserved	21	S_USER1	93	VCC3	143
PC_INT	122	Reserved	28	S_USER2	95	VCC3	165
PCI_DET	167	Reserved	31	SP0SEL	12	VCC3	191
PCI_FLT	64	Reserved	33	SW_RST	120	VCC5	74
PCI_INT	131	Reserved	39	TBC_INT	201	VCC5	83
PCI_IRQ	4	Reserved	45	TBC_RD	121	VCC5	179
PCI_RST	153	Reserved	57	TBC_RDY	205	VCC5	186
PCIMRD_INT	42	Reserved	66	TBC_RST	44	WRFIFO	133
PCIMWR_INT	61	Reserved	67	TBC_WR	187	WRFULL	172
PT_ADR	197	Reserved	87	TBCA0_HBE0	194	XCNTL0	6
PT_RDY	113	Reserved	88	TBCA1_HBE1	130	XCNTL1	15
PTATN	154	Reserved	91	TBCA2_HHWIL	81	XRDY	160
PTBE0	25	Reserved	114	TBCA3_HCNTL0	192	XSTAT0	98
PTBE1	195	Reserved	163	TBCA4_HCNTL1	188	XSTAT1	112

C.2 EVM CPLD Equations

The following listing provides a table of contents to the CPLD's source files included in this section.

```
MODULE EVM

TITLE 'TMS320C6x EVM Top-Level Programmable Logic'

"DWG NAME  TMS320C6x Evaluation Module (EVM)
"ASSY#     D600830-0001
"PAL #     U12, 830*
"PAL TYPE  Altera EPM7256SQC208-10 (256-Macrocells, 10-ns, 208-PQFP)
"COMPANY   Texas Instruments Incorporated / DNA Enterprises, Inc.
"ENGR      Brian G. Carlson
"DATE      02/21/98
"TOOLS     Synario ABEL 6.5 / Altera Max Plus II 8.2

"DESCRIPTION
   "This is the top-level module for the TMS320C6x EVM's CPLD.  It
   "defines the CPLD's pin names, the low-level modules that are used,
   "and all inter-module connections.  All CPLD registers that are
   "memory-mapped into the PCI and DSP memory spaces are implemented
   "in this top-level module.  The ability to float all CPLD output
   "pins to support in-circuit testing is also implemented in this module.


"REVISIONS
"   04/15/98 - 1. Changed CPLD_REV to 3.
"              2. Added host/DSP semaphores.
"              3. Added OSC B frequency register
"
"   03/11/98 - Changed CPLD_REV to 2
"
"   03/05/98 - Changed CPLD_REV to 1

"Sub-module prototypes
   EVMRESET interface  (BRD_RST_,SW_RST_,VCC2BAD_,PCI_RST_,PCI_DET_,
                        xreset,dsprst,tbcrst ->
                        MAN_RST_,DSP_RST_,TBC_RST_,EXT_RST_);

   EVMSWMUX interface  (h_clkmode,h_clksel,h_endian,h_jtagsel,h_user2..h_user0,
                        swsel,h_bootmode4..h_bootmode0,S_CLKMODE,S_CLKSEL,
                        S_ENDIAN,S_JTAGSEL,S_USER2..S_USER0,
                        S_BOOTMODE4..S_BOOTMODE0 ->
                        BOOTMODE4..BOOTMODE0,CLKMODE,LENDIAN,USER2..USER0,
                        clksel,JTAGSEL);

   EVMCKSEL interface  (CLKA_DIV2,clksel -> osc_a_, osc_b_);
```

```
   EVMDECOD interface  (RESET,PCI_DET_,PCICLK,DSPCLK,
                        CE1_,CE2_,CE3_,ARE_,AWE_,EA21..EA16,
                        ao_bsy,emif_ack,XRDY,ce2sden,ce3sden,CLKMODE –>
                        DSP2AOD_,DSP2PBD_,DSP2XD_,are_r_,awe_r_,emif_req,
                        CPLD_CS,CODEC_CS_,CODEC_WR_,CODEC_RD_,ARDY);
   EVMINT   interface  (RESET,DSP_HINT_,TBC_INT_,PCICLK,PCI_DET_,PCI_IRQ_,
                        RDEMPTY,WRFULL,CODEC_IRQ, hinten, tbcinten,
                        dspnmi,nmisel,nmien,pcimren,pcimwen,RDFIFO_,WRFIFO_ –>
                        PC_INT,NMI,PCI_INT,PCIMRD_INT,PCIMWR_INT);
   EVMPCI   interface  (RESET,ARE_,AWE_,BE3_,BE2_,BE1_,BE0_,EA16,EA6..EA2,
                        PCI_DET_,PCICLK,PTATN_,PTBURST_,PTNUM1,PTNUM0,PTWR,
                        pt_be3_,pt_be2_,pt_be1_,pt_be0_,
                        pt_addr4,pt_addr3,pt_addr2,pt_addr1, pt_addr0,
                        TBC_RDY_,DSP_HRDY_,emif_req –>
                        PCI_FLT_,PT_ADR_,PT_RDY_,AO_SEL_,AO_WR_,AO_RD_,
                        AO_ADR6..AO_ADR2,AO_BE3_,AO_BE2_,AO_BE1_,AO_BE0_,RDFIFO_,WRFIFO_,
                        pcireg_oe,pcireg_ce,TBC_WR_,TBC_RD_,
                        HCS_,HDS1_,HRW,TBCA4_HCNTL1,TBCA3_HCNTL0,TBCA2_HHWIL,
                        TBCA1_HBE1_,TBCA0_HBE0_,ao_bsy,emif_ack);

"Sub–module instantiations
   RESET     functional_block EVMRESET;    "Board, DSP, TBC and daughterboard resets
   SW_MUX    functional_block EVMSWMUX;    "HW/SW switch selection mux
   CLOCK_SEL functional_block EVMCKSEL;    "DSP clock source selection mux
   DECODE    functional_block EVMDECOD;    "Address decode logic and device control
   INT       functional_block EVMINT;      "Interrupt control
   PCI       functional_block EVMPCI;      "PCI interface control

"INPUTS
   "NOTE: Active low signals are indicated with trailing underscores.
   "–––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––
   "Float Outputs Test Input
      FLOAT_       pin;      "Floats all outputs to support in–circuit testing
   "–––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––
   "Resets
      BRD_RST_     pin;      "Reset from voltage supervisor
      SW_RST_      pin;      "Reset from push button switch
      VCC2BAD_     pin;      "VCC bad from the power supervisor
      PCI_RST_     pin;      "PCI system reset from PCI controller
   "–––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––
   "Clocks
      DSPCLK       pin;      "DSP's CLKOUT2
      CLKA_DIV2    pin;      "DSP clock source (OSC A) divided by 2
```

```
"-------------------------------------------------------------------------------

"From TMS320C6201 DSP
   CE1_          pin;      "EMIF CE1 memory space enable
   CE2_          pin;      "EMIF CE2 memory space enable
   CE3_          pin;      "EMIF CE3 memory space enable
   ARE_          pin;      "EMIF asynchronous memory read strobe
   AWE_          pin;      "EMIF asynchronous memory write strobe
   BE3_          pin;      "EMIF byte 3 enable control
   BE2_          pin;      "EMIF byte 2 enable control
   BE1_          pin;      "EMIF byte 1 enable control
   BE0_          pin;      "EMIF byte 0 enable control
   BE = [BE3_, BE2_, BE1_, BE0_];

   "EMIF word address bits [21..16], [6..2]
   EA21          pin;
   EA20          pin;
   EA19          pin;
   EA18          pin;
   EA17          pin;
   EA16          pin;
   EA6           pin;
   EA5           pin;
   EA4           pin;
   EA3           pin;
   EA2           pin;

   "High address is for device selection
   EA_HI  = [EA21, EA20, EA19, EA18, EA17, EA16];

   "Low address is for register selection
   EA_LO  = [EA6,  EA5,  EA4,  EA3,  EA2];

   DSP_HRDY_     pin;      "DSP HPI ready
   DSP_HINT_     pin;      "DSP HPI host interrupt

   DSP_PD        pin;      "DSP power down indication

"-------------------------------------------------------------------------------

"From S5933 PCI Controller
   BPCLK         pin;      "Buffered PCI clock (33 MHz max.)
   PTATN_        pin;      "Pass-thru attention indication
   PTBURST_      pin;      "Pass-thru burst indication
   PTNUM1        pin;      "Pass-thru region number (bit 1)
   PTNUM0        pin;      "Pass-thru region number (bit 0)
   PTWR          pin;      "Pass-thru access type (R=0, W=1)
   PCI_IRQ_      pin;      "Interrupt to add-on device
```

```
   PTBE3_       pin;      "Pass-thru byte enable 3
   PTBE2_       pin;      "Pass-thru byte enable 2
   PTBE1_       pin;      "Pass-thru byte enable 1
   PTBE0_       pin;      "Pass-thru byte enable 0
   PCI_DET_     pin;      "PCI bus detection (0=PCI, 1=Standalone)
   RDEMPTY      pin;      "Read FIFO empty flag
   WRFULL       pin;      "Write FIFO full flag

"-------------------------------------------------------------------------------

"From JTAG TBC
   TBC_RDY_     pin;      "TBC ready
   TBC_INT_     pin;      "TBC interrupt

"-------------------------------------------------------------------------------

"From DIP Switch

   "Boot mode selection for No-boot, HPI-boot or ROM-boot
   S_BOOTMODE4  pin;      "Boot mode 4 (SW2-1)
   S_BOOTMODE3  pin;      "Boot mode 3 (SW2-2)
   S_BOOTMODE2  pin;      "Boot mode 2 (SW2-3)
   S_BOOTMODE1  pin;      "Boot mode 1 (SW2-4)
   S_BOOTMODE0  pin;      "Boot mode 0 (SW2-5)
   S_BOOTMODE = [S_BOOTMODE4, S_BOOTMODE3, S_BOOTMODE2, S_BOOTMODE1, S_BOOTMODE0];

   S_CLKMODE    pin;      "Selects x1 (No PLL) or x4 (PLL) Clock Mode (SW2-6)

   S_CLKSEL     pin;      "Selects output of oscillator A or B (SW2-7)
   S_ENDIAN     pin;      "Selects big or little endian memory addressing (SW2-8)

   S_JTAGSEL    pin;      "Selects internal or external JTAG emulation  (SW2-9)
   S_USER2      pin;      "User-defined switch (S2-10)
   S_USER1      pin;      "User-defined switch (S2-11)
   S_USER0      pin;      "User-defined switch (S2-12)

"-------------------------------------------------------------------------------

"From Daughterboard (Expansion Peripheral Interface)
   XSTAT1       pin;      "External status 1
   XSTAT0       pin;      "External status 0
   DB_INT       pin;      "Daughterboard interrupt (EXT_INT7)
   XRDY         pin;      "External asynchronous memory access ready

"-------------------------------------------------------------------------------

"From CS4231A Audio Codec
   CODEC_IRQ    pin;      "Audio codec interrupt
   CODEC_PD_    pin;      "Audio codec power down indication
```

```
"BIDIR
   "-------------------------------------------------------------------------------

   "From TMS320C6201 DSP

      "EMIF data bits [7..0]
      ED7          pin;
      ED6          pin;
      ED5          pin;
      ED4          pin;
      ED3          pin;
      ED2          pin;
      ED1          pin;
      ED0          pin;
      ED = [ED7, ED6, ED5, ED4, ED3, ED2, ED1, ED0];

   "-------------------------------------------------------------------------------

   "From S5933 PCI Controller
      "Add-on data bits [7..0]
      DQ7          pin;
      DQ6          pin;
      DQ5          pin;
      DQ4          pin;
      DQ3          pin;
      DQ2          pin;
      DQ1          pin;
      DQ0          pin;
      DQ = [DQ7..DQ0];

"OUTPUTS
   "-------------------------------------------------------------------------------

   "To Data Buffers/Transceivers
      DSP2AOD_     pin istype 'com';   "Connects DSP to add-on bus
      DSP2PBD_     pin istype 'com';   "Connects DSP to peripheral bus
      DSP2XD_      pin istype 'com';   "Connects DSP to external bus

   "-------------------------------------------------------------------------------

   "To JTAG TBC and DSP HPI
      TBCA0_HBE0_  pin istype 'com';   "TBC address bit 0 / HPI byte enable 0
      TBCA1_HBE1_  pin istype 'com';   "TBC address bit 1 / HPI byte enable 1
      TBCA2_HHWIL  pin istype 'com';   "TBC address bit 2 / HPI half-word select
      TBCA3_HCNTL0 pin istype 'com';   "TBC address bit 3 / HPI control 0
      TBCA4_HCNTL1 pin istype 'com';   "TBC address bit 4 / HPI control 1
```

```
"-------------------------------------------------------------------------------

"To HPI
   HCS_          pin istype 'com';   "HPI chip select
   HDS1_         pin istype 'com';   "HPI data strobe 1
   HRW           pin istype 'com'    "HPI read (1) / write (0) select

"-------------------------------------------------------------------------------

"To JTAG TBC
   TBC_WR_       pin istype 'com';   "TBC write strobe
   TBC_RD_       pin istype 'com';   "TBC read strobe
   TBC_RST_      pin istype 'com';   "TBC hardware reset

"-------------------------------------------------------------------------------

"To S5933 PCI Controller
   PCI_FLT_      pin istype 'com';   "Float PCI controller outputs
   PT_ADR_       pin istype 'com';   "Pass-thru address request
   PT_RDY_       pin istype 'com';   "Pass-thru ready indication
   AO_SEL_       pin istype 'com';   "Add-on select for register access
   AO_WR_        pin istype 'com';   "Add-on write strobe
   AO_RD_        pin istype 'com';   "Add-on read strobe

   "Add-on address bits [6..2]
   AO_ADR6       pin istype 'com';
   AO_ADR5       pin istype 'com';
   AO_ADR4       pin istype 'com';
   AO_ADR3       pin istype 'com';
   AO_ADR2       pin istype 'com';
   AO_ADR = [AO_ADR6..AO_ADR2];

   "Add-on byte enables [3..0]
   AO_BE3_       pin istype 'com';
   AO_BE2_       pin istype 'com';
   AO_BE1_       pin istype 'com';
   AO_BE0_       pin istype 'com';

   RDFIFO_       pin istype 'com';   "Read FIFO strobe
   WRFIFO_       pin istype 'com';   "Write FIFO strobe

   PC_INT        pin istype 'com';      "Add-on-to-PCI interrupt

"-------------------------------------------------------------------------------
```

```
"To TMS320C6201 DSP
   "Boot mode selection for No-boot, HPI-boot or ROM-boot
   BOOTMODE4    pin istype 'com';
   BOOTMODE3    pin istype 'com';
   BOOTMODE2    pin istype 'com';
   BOOTMODE1    pin istype 'com';
   BOOTMODE0    pin istype 'com';

   CLKMODE      pin istype 'com';   "Clock mode
   LENDIAN      pin istype 'com';   "Little-endian selection
   NMI          pin istype 'com';   "NMI interrupt from codec or host
   PCI_INT      pin istype 'com';   "Interrupt from PCI controller (EXT_INT4)
   PCIMRD_INT   pin istype 'com';   "PCI master read interrupt (EXT_INT5)
   PCIMWR_INT   pin istype 'com';   "PCI master write interrupt (EXT_INT6)
   DSP_RST_     pin istype 'com';   "DSP reset
   ARDY         pin istype 'com';   "EMIF asynchronous memory access ready

"-------------------------------------------------------------------------------

"To Multiplexers
   OSC_A_EN_    pin istype 'com';   "DSP oscillator A enable
   OSC_B_EN_    pin istype 'com';   "DSP oscillator B enable

   JTAGSEL      pin istype 'com';   "JTAG selection (int/ext)

   SP0SEL       pin istype 'reg_d, dc, buffer';  "McBSP0 selection

"-------------------------------------------------------------------------------

"To User LEDs
   LED1_        pin istype 'reg_d, dc, buffer';  "User LED 1 control
   LED0_        pin istype 'reg_d, dc, buffer';  "User LED 0 control

"-------------------------------------------------------------------------------

"To Daughterboard (Expansion Peripheral Interface)
   XCNTL1       pin istype 'reg_d, dc, buffer'; "External control 1
   XCNTL0       pin istype 'reg_d, dc, buffer'; "External control 0

   EXT_RST_     pin istype 'com';   "External reset

"-------------------------------------------------------------------------------

"To CS4231A Audio Codec
   CODEC_CS_    pin istype 'com';   "Codec chip select
   CODEC_WR_    pin istype 'com';   "Codec write strobe
   CODEC_RD_    pin istype 'com';   "Codec read strobe

"-------------------------------------------------------------------------------
```

```
   "To SDRAMs
      CE3_SDEN     pin istype 'com';   "Bank 1 (CE3) SDRAM enable (CKE)
      CE2_SDEN     pin istype 'com';   "Bank 0 (CE2) SDRAM enable (CKE)

   "-------------------------------------------------------------------------------

   "To Voltage Supervisor
      MAN_RST_     pin istype 'com';   "Pushbutton or PCI system reset

"-------------------------------------------------------------------------------

"Internal Nodes
      CPLD_CS      node istype 'com';  "CPLD DSP register chip select

"-------------------------------------------------------------------------------

"Internal Registers

   "DSP-mapped register bits
      xreset       node istype 'reg_d, dc, buffer';  "External (DB) reset
      nmisel       node istype 'reg_d, dc, buffer';  "NMI selection (codec/host)
      nmien        node istype 'reg_d, dc, buffer';  "NMI interrupt enable
      pcimwen      node istype 'reg_d, dc, buffer';  "PCI master write enable
      pcimren      node istype 'reg_d, dc, buffer';  "PCI master read enable
      ce2sden      node istype 'reg_d, dc, buffer';  "CE2 SDRAM enable
      ce3sden      node istype 'reg_d, dc, buffer';  "CE3 SDRAM enable
      dspsem0      node istype 'reg_d, dc, buffer';  "DSP semaphore 0
      dspsem1      node istype 'reg_d, dc, buffer';  "DSP semaphore 1

   "PCI-mapped register bits
      dspnmi       node istype 'reg_d, dc, buffer';  "Host NMI to DSP
      tbcinten     node istype 'reg_d, dc, buffer';  "Host TBC interrupt enable
      hinten       node istype 'reg_d, dc, buffer';  "Host DSP host interrupt enable
      tbcrst       node istype 'reg_d, dc, buffer';  "Host TBC reset
      dsprst       node istype 'reg_d, dc, buffer';  "Host DSP reset
      h_clkmode    node istype 'reg_d, dc, buffer';  "Clock mode software switch
      h_clksel     node istype 'reg_d, dc, buffer';  "Clock selection software switch
      h_endian     node istype 'reg_d, dc, buffer';  "Endian software switch
      h_jtagsel    node istype 'reg_d, dc, buffer';  "JTAG selection software switch
      h_user2      node istype 'reg_d, dc, buffer';  "User-defined software switch 2
      h_user1      node istype 'reg_d, dc, buffer';  "User-defined software switch 1
      h_user0      node istype 'reg_d, dc, buffer';  "User-defined software switch 0
      swsel        node istype 'reg_d, dc, buffer';  "DIP or software switch selection
      h_bootmode4  node istype 'reg_d, dc, buffer';  "Host boot mode selection [4..0]
      h_bootmode3  node istype 'reg_d, dc, buffer';
      h_bootmode2  node istype 'reg_d, dc, buffer';
      h_bootmode1  node istype 'reg_d, dc, buffer';
      h_bootmode0  node istype 'reg_d, dc, buffer';
```

```
      h_bootmode = [h_bootmode4..h_bootmode0] ;
      hostsem0     node istype 'reg_d, dc, buffer';  "Host semaphore 0
      hostsem1     node istype 'reg_d, dc, buffer';  "Host semaphore 1

   "Registered pass-thru address used to drive address to TBC, CPLD REG and HPI.
      pt_addr4     node istype 'reg_d, dc, buffer';
      pt_addr3     node istype 'reg_d, dc, buffer';
      pt_addr2     node istype 'reg_d, dc, buffer';
      pt_addr1     node istype 'reg_d, dc, buffer';
      pt_addr0     node istype 'reg_d, dc, buffer';
      pt_addr = [pt_addr4..pt_addr0] ;

   "Registered pass-thru byte enables used to drive HPI BE's.
      pt_be3_      node istype 'reg_d, dc, buffer';
      pt_be2_      node istype 'reg_d, dc, buffer';
      pt_be1_      node istype 'reg_d, dc, buffer';
      pt_be0_      node istype 'reg_d, dc, buffer';
      pt_be_  = [pt_be3_,pt_be2_,pt_be1_,pt_be0_];

"Constants

   h,l,x = 1, 0, .X. ;

   "Revision Control
      CPLD_REV   = ^h03;

   "Reserved Register Values
      PCI_RSVD   = ^h00;
      DSP_RSVD   = ^h00;

   "Oscillator B Frequency
      OSC_B_FREQ = 0;            "0 defaults to 40 MHz

   "Global reset
      reset = !BRD_RST_;         "Internal active-high reset alias


   "-------------------------------------------------------------------------------
   "DSP-Mapped Registers
   "-------------------------------------------------------------------------------

      "Register Offsets [EA6..EA2]       MAP0          MAP1
         DSP_CNTL_ADR     = 0;   "CE1:0x0138xxx0 / 0x0178xxx0
         DSP_STAT_ADR     = 1;   "CE1:0x0138xxx4 / 0x0178xxx4
         DSP_DIPOPT_ADR   = 2;   "CE1:0x0138xxx8 / 0x0178xxx8
         DSP_DIPBOOT_ADR  = 3;   "CE1:0x0138xxxC / 0x0178xxxC
         DSP_DSPOPT_ADR   = 4;   "CE1:0x0138xx10 / 0x0178xx10
```

```
        DSP_DSPBOOT_ADR  = 5;   "CE1:0x0138xx14 / 0x0178xx14
        DSP_FIFOSTAT_ADR = 6;   "CE1:0x0138xx18 / 0x0178xx18
        DSP_SDCNTL_ADR   = 7;   "CE1:0x0138xx1C / 0x0178xx1C
        DSP_OSCB_ADR     = 8;   "CE1:0x0138xx20 / 0x0178xx20
        DSP_SEM0_ADR     = 9;   "CE1:0x0138xx24 / 0x0178xx24
        DSP_SEM1_ADR     = 10;  "CE1:0x0138xx28 / 0x0178xx28

    "-------------------------------------------------------------------------------
    "Bit Definitions
    "-------------------------------------------------------------------------------

        dsp_cntl   = [XCNTL1, XCNTL0,  xreset, nmien, nmisel, SP0SEL, !LED1_, !LED0_];
        "             RW        RW       RW     RW      RW      RW      RW     RW

        dsp_stat   = [XSTAT1, XSTAT0, DB_INT, dspnmi,  !CODEC_PD_,  CODEC_IRQ, !PCI_IRQ_, !PCI_DET_];
        "             R       R        R       R          R           R          R          R

        dsp_dipopt  = [0, S_CLKMODE, S_CLKSEL, S_ENDIAN, S_JTAGSEL, S_USER2, S_USER1, S_USER0];
        "             R     R           R         R          R         R       R         R

        dsp_dipboot = [0, 0, 0, S_BOOTMODE4, S_BOOTMODE3, S_BOOTMODE2, S_BOOTMODE1, S_BOOTMODE0];
        "              R  R  R       R            R            R            R            R

        dsp_dspopt = [0, CLKMODE, SW_MUX.clksel, LENDIAN, JTAGSEL, SW_MUX.USER2, SW_MUX.USER1,
SW_MUX.USER0];
        "             R     R         R         R        R       R       R             R

        dsp_dspboot = [swsel, 0, 0, BOOTMODE4, BOOTMODE3, BOOTMODE2, BOOTMODE1, BOOTMODE0];
        "               R    R  R      R          R          R          R          R

        dsp_fifostat= [0, 0, PCIMRD_INT, PCIMWR_INT, RDEMPTY, WRFULL, pcimren, pcimwen];
        "             R  R        R           R         R       R       RW       RW

        dsp_sdcntl = [0, 0, 0, 0, 0, 0, ce3sden, ce2sden];
        "             R  R  R  R  R  R     RW       RW

        dsp_sem0    = [0, 0, 0, 0, 0, 0, 0, dspsem0];
        "              R  R  R  R  R  R  R    RW

        dsp_sem1    = [0, 0, 0, 0, 0, 0, 0, dspsem1];
        "              R  R  R  R  R  R  R    RW

    "-------------------------------------------------------------------------------
    "Clock enables for DSP register writes
    "-------------------------------------------------------------------------------

       "CPLD_CS is active when EMIF access to 0x0138xxxx (MAP0)/0x0178xxxx (MAP1)
       "EA[6..2] select register offset
```

```
      dsp_cntl_ce      = CPLD_CS & (EA_LO == DSP_CNTL_ADR);
      dsp_stat_ce      = CPLD_CS & (EA_LO == DSP_STAT_ADR);
      dsp_dipopt_ce    = CPLD_CS & (EA_LO == DSP_DIPOPT_ADR);
      dsp_dipboot_ce   = CPLD_CS & (EA_LO == DSP_DIPBOOT_ADR);
      dsp_dspopt_ce    = CPLD_CS & (EA_LO == DSP_DSPOPT_ADR);
      dsp_dspboot_ce   = CPLD_CS & (EA_LO == DSP_DSPBOOT_ADR);
      dsp_fifostat_ce  = CPLD_CS & (EA_LO == DSP_FIFOSTAT_ADR);
      dsp_sdcntl_ce    = CPLD_CS & (EA_LO == DSP_SDCNTL_ADR);
      dsp_sem0_ce      = CPLD_CS & (EA_LO == DSP_SEM0_ADR);
      dsp_sem1_ce      = CPLD_CS & (EA_LO == DSP_SEM1_ADR);

"-------------------------------------------------------------------------------
"PCI-Mapped Registers
"-------------------------------------------------------------------------------

   "Register offsets [A6..A2]
      PCI_CNTL_ADR     = 0;   "BAR2+0x00
      PCI_STAT_ADR     = 1;   "BAR2+0x04
      PCI_SWOPT_ADR    = 2;   "BAR2+0x08
      PCI_SWBOOT_ADR   = 3;   "BAR2+0x0C
      PCI_DIPOPT_ADR   = 4;   "BAR2+0x10
      PCI_DIPBOOT_ADR  = 5;   "BAR2+0x14
      PCI_DSPOPT_ADR   = 6;   "BAR2+0x18
      PCI_DSPBOOT_ADR  = 7;   "BAR2+0x1C
      PCI_CPLDREV_ADR  = 8;   "BAR2+0x20
      PCI_SEM0_ADR     = 9;   "BAR2+0x24
      PCI_SEM1_ADR     = 10;  "BAR2+0x28

"-------------------------------------------------------------------------------

   "Register bit definitions
      pci_cntl    = [dspnmi, tbcinten, hinten, !TBC_RDY_, !TBC_INT_, !DSP_HINT_, tbcrst,  dsprst];
      "              RW       RW        RW       R          R          R          RW       RW

      pci_stat    = [0, !VCC2BAD_, !CODEC_PD_, DSP_PD, XCNTL1, XCNTL0, !LED1_, !LED0_];
      "                R           R          R        R       R       R      R

      pci_swopt   = [0, h_clkmode, h_clksel, h_endian, h_jtagsel, h_user2, h_user1, h_user0];
      "               R      RW        RW         RW         RW         RW       RW       RW

      pci_swboot  = [swsel, 0, 0, h_bootmode4, h_bootmode3, h_bootmode2, h_bootmode1, h_bootmode0];
      "                RW   R  R       RW           RW           RW           RW           RW

      pci_dipopt  = [0, S_CLKMODE, S_CLKSEL, S_ENDIAN, S_JTAGSEL, S_USER2, S_USER1, S_USER0];
      "               R      R          R         R          R         R        R        R

      pci_dipboot = [0, 0, 0, S_BOOTMODE4, S_BOOTMODE3, S_BOOTMODE2, S_BOOTMODE1, S_BOOTMODE0];
      "               R  R  R       R            R            R            R            R
```

```
            pci_dspopt  = [0, CLKMODE, SW_MUX.clksel, LENDIAN, JTAGSEL, SW_MUX.USER2, SW_MUX.USER1,
SW_MUX.USER0];
            ″              R     R        R              R       R          R             R             R

            pci_dspboot = [0, 0, 0, BOOTMODE4, BOOTMODE3, BOOTMODE2, BOOTMODE1, BOOTMODE0];
            ″               R  R  R     R          R          R          R          R

            pci_cpldrev = CPLD_REV;

            pci_sem0    = [0, 0, 0, 0, 0, 0, 0, hostsem0];
            ″               R  R  R  R  R  R  R     RW

            pci_sem1    = [0, 0, 0, 0, 0, 0, 0, hostsem1];
            ″               R  R  R  R  R  R  R     RW

            pci_reserve = PCI_RSVD;

      ″-------------------------------------------------------------------------------

         ″Clock enables for PCI register writes
            pci_cntl_ce   = PCI.pcireg_ce & (pt_addr == PCI_CNTL_ADR);
            pci_swopt_ce  = PCI.pcireg_ce & (pt_addr == PCI_SWOPT_ADR);
            pci_swboot_ce = PCI.pcireg_ce & (pt_addr == PCI_SWBOOT_ADR);
            pci_sem0_ce   = PCI.pcireg_ce & (pt_addr == PCI_SEM0_ADR);
            pci_sem1_ce   = PCI.pcireg_ce & (pt_addr == PCI_SEM1_ADR);

Equations

      ″=============================================================================
      ″   ABEL Low-level Module Connections
      ″=============================================================================

      ″The following several sections define the connections between this
      ″top-level ABEL source module and the low-level modules, as well as
      ″connections between low-level modules.  These connections are similar
      ″to a high-level block diagram definition between modules.

      ″-------------------------------------------------------------------------------
      ″RESET I/O Connections
      ″-------------------------------------------------------------------------------

      ″Reset Inputs
         RESET.BRD_RST_ = BRD_RST_;
         RESET.SW_RST_  = SW_RST_;
         RESET.VCC2BAD_ = VCC2BAD_;
         RESET.PCI_RST_ = PCI_RST_;
         RESET.PCI_DET_ = PCI_DET_;
         RESET.xreset   = xreset;
         RESET.dsprst   = dsprst;
         RESET.tbcrst   = tbcrst;
```

```
"Reset Outputs
   MAN_RST_ = RESET.MAN_RST_;
   DSP_RST_ = RESET.DSP_RST_;
   TBC_RST_ = RESET.TBC_RST_;
   EXT_RST_ = RESET.EXT_RST_;
"-------------------------------------------------------------------------------
"SW_MUX I/O Connections
"-------------------------------------------------------------------------------
"Switch Multiplexer Inputs
   SW_MUX.h_clkmode    = h_clkmode;
   SW_MUX.h_clksel     = h_clksel;
   SW_MUX.h_endian     = h_endian;
   SW_MUX.h_jtagsel    = h_jtagsel;
   SW_MUX.h_user2      = h_user2;
   SW_MUX.h_user1      = h_user1;
   SW_MUX.h_user0      = h_user0;
   SW_MUX.swsel        = swsel;
   SW_MUX.h_bootmode4  = h_bootmode4;
   SW_MUX.h_bootmode3  = h_bootmode3;
   SW_MUX.h_bootmode2  = h_bootmode2;
   SW_MUX.h_bootmode1  = h_bootmode1;
   SW_MUX.h_bootmode0  = h_bootmode0;
   SW_MUX.S_CLKMODE    = S_CLKMODE;
   SW_MUX.S_CLKSEL     = S_CLKSEL;
   SW_MUX.S_ENDIAN     = S_ENDIAN;
   SW_MUX.S_JTAGSEL    = S_JTAGSEL;
   SW_MUX.S_USER2      = S_USER2;
   SW_MUX.S_USER1      = S_USER1;
   SW_MUX.S_USER0      = S_USER0;
   SW_MUX.S_BOOTMODE4  = S_BOOTMODE4;
   SW_MUX.S_BOOTMODE3  = S_BOOTMODE3;
   SW_MUX.S_BOOTMODE2  = S_BOOTMODE2;
   SW_MUX.S_BOOTMODE1  = S_BOOTMODE1;
   SW_MUX.S_BOOTMODE0  = S_BOOTMODE0;
"Switch Multiplexer Outputs
   BOOTMODE4 = SW_MUX.BOOTMODE4;
   BOOTMODE3 = SW_MUX.BOOTMODE3;
   BOOTMODE2 = SW_MUX.BOOTMODE2;
   BOOTMODE1 = SW_MUX.BOOTMODE1;
   BOOTMODE0 = SW_MUX.BOOTMODE0;
   CLKMODE   = SW_MUX.CLKMODE;
   LENDIAN   = SW_MUX.LENDIAN;
   JTAGSEL   = SW_MUX.JTAGSEL & !PCI_DET_; "force ext JTAG when not in PCI slot
```

```
"-------------------------------------------------------------------------------
"CLOCK_SEL I/O Connections
"-------------------------------------------------------------------------------

"Clock Select Inputs
    CLOCK_SEL.CLKA_DIV2 = CLKA_DIV2;
    CLOCK_SEL.clksel    = SW_MUX.clksel;

"Clock Select Outputs
    OSC_A_EN_ = CLOCK_SEL.osc_a_;
    OSC_B_EN_ = CLOCK_SEL.osc_b_;

"-------------------------------------------------------------------------------
"DECODE I/O Connections
"-------------------------------------------------------------------------------

"Decode Inputs
    DECODE.RESET     = reset;
    DECODE.PCI_DET_  = PCI_DET_;
    DECODE.PCICLK    = BPCLK;
    DECODE.DSPCLK    = DSPCLK;
    DECODE.CE1_      = CE1_;
    DECODE.CE2_      = CE2_;
    DECODE.CE3_      = CE3_;
    DECODE.ARE_      = ARE_;
    DECODE.AWE_      = AWE_;
    DECODE.EA21      = EA21;
    DECODE.EA20      = EA20;
    DECODE.EA19      = EA19;
    DECODE.EA18      = EA18;
    DECODE.EA17      = EA17;
    DECODE.EA16      = EA16;
    DECODE.ao_bsy    = PCI.ao_bsy;
    DECODE.emif_ack  = PCI.emif_ack;
    DECODE.XRDY      = XRDY;
    DECODE.ce2sden   = ce2sden;
    DECODE.ce3sden   = ce3sden;
    DECODE.CLKMODE   = SW_MUX.CLKMODE;

"Decode Outputs
    DSP2AOD_  = DECODE.DSP2AOD_;
    DSP2PBD_  = DECODE.DSP2PBD_;
    DSP2XD_   = DECODE.DSP2XD_;
    CPLD_CS   = DECODE.CPLD_CS;
    CODEC_CS_ = DECODE.CODEC_CS_;
    CODEC_WR_ = DECODE.CODEC_WR_;
```

```
    CODEC_RD_ = DECODE.CODEC_RD_;
    ARDY      = DECODE.ARDY;

"-------------------------------------------------------------------------------
"INT I/O Connections
"-------------------------------------------------------------------------------

"Interrupt Inputs
    INT.RESET    = reset;
    INT.DSP_HINT_= DSP_HINT_;
    INT.TBC_INT_ = TBC_INT_;
    INT.PCICLK   = BPCLK;
    INT.PCI_DET_ = PCI_DET_;

    INT.PCI_IRQ_ = PCI_IRQ_;
    INT.RDEMPTY  = RDEMPTY;
    INT.WRFULL   = WRFULL;
    INT.CODEC_IRQ= CODEC_IRQ;
    INT.hinten   = hinten;
    INT.tbcinten = tbcinten;
    INT.dspnmi   = dspnmi;
    INT.nmisel   = nmisel;
    INT.nmien    = nmien;
    INT.pcimren  = pcimren;
    INT.pcimwen  = pcimwen;
    INT.RDFIFO_  = PCI.RDFIFO_;
    INT.WRFIFO_  = PCI.WRFIFO_;

"Interrupt Outputs
    PC_INT       = INT.PC_INT;
    NMI          = INT.NMI;
    PCI_INT      = INT.PCI_INT;
    PCIMRD_INT   = INT.PCIMRD_INT;
    PCIMWR_INT   = INT.PCIMWR_INT;

"-------------------------------------------------------------------------------
"PCI I/O Connections
"-------------------------------------------------------------------------------

"PCI Control Inputs
PCI.RESET    = reset;
PCI.ARE_     = DECODE.are_r_;
PCI.AWE_     = DECODE.awe_r_;
PCI.BE3_     = BE3_;
PCI.BE2_     = BE2_;
PCI.BE1_     = BE1_;
PCI.BE0_     = BE0_;
```

```
PCI.EA16     = EA16;
PCI.EA6      = EA6;
PCI.EA5      = EA5;
PCI.EA4      = EA4;
PCI.EA3      = EA3;
PCI.EA2      = EA2;
PCI.PCI_DET_ = PCI_DET_;
PCI.PCICLK   = BPCLK;
PCI.PTATN_   = PTATN_;
PCI.PTBURST_ = PTBURST_;
PCI.PTNUM1   = PTNUM1;
PCI.PTNUM0   = PTNUM0;
PCI.PTWR     = PTWR;

PCI.pt_be3_  = pt_be3_;
PCI.pt_be2_  = pt_be2_;
PCI.pt_be1_  = pt_be1_;
PCI.pt_be0_  = pt_be0_;
PCI.pt_addr4 = pt_addr4;
PCI.pt_addr3 = pt_addr3;
PCI.pt_addr2 = pt_addr2;
PCI.pt_addr1 = pt_addr1;
PCI.pt_addr0 = pt_addr0;
PCI.TBC_RDY_ = TBC_RDY_;
PCI.DSP_HRDY_= DSP_HRDY_;
PCI.emif_req = DECODE.emif_req;

"PCI Control Outputs
PCI_FLT_     = PCI.PCI_FLT_;
PT_ADR_      = PCI.PT_ADR_;
PT_RDY_      = PCI.PT_RDY_;
AO_SEL_      = PCI.AO_SEL_;
AO_WR_       = PCI.AO_WR_;
AO_RD_       = PCI.AO_RD_;
AO_ADR6      = PCI.AO_ADR6;
AO_ADR5      = PCI.AO_ADR5;
AO_ADR4      = PCI.AO_ADR4;
AO_ADR3      = PCI.AO_ADR3;
AO_ADR2      = PCI.AO_ADR2;
AO_BE3_      = PCI.AO_BE3_;
AO_BE2_      = PCI.AO_BE2_;
AO_BE1_      = PCI.AO_BE1_;
AO_BE0_      = PCI.AO_BE0_;
RDFIFO_      = PCI.RDFIFO_;
WRFIFO_      = PCI.WRFIFO_;
```

```
TBC_WR_       = PCI.TBC_WR_;
TBC_RD_       = PCI.TBC_RD_;
HCS_          = PCI.HCS_;
HDS1_         = PCI.HDS1_;
HRW           = PCI.HRW;
TBCA4_HCNTL1  = PCI.TBCA4_HCNTL1;
TBCA3_HCNTL0  = PCI.TBCA3_HCNTL0;
TBCA2_HHWIL   = PCI.TBCA2_HHWIL;
TBCA1_HBE1_   = PCI.TBCA1_HBE1_;
TBCA0_HBE0_   = PCI.TBCA0_HBE0_;

"=============================================================================
"   Latch Pass-Thru Address and Byte Enables
"=============================================================================

"Pass-thru address latch
"Address is latched on rising edge of PCI clock when PT_ADR_ is asserted
   pt_addr.ap    = reset;
   pt_addr.clk   = BPCLK;
   pt_addr.ce    = !PT_ADR_;
   pt_addr      := [DQ6, DQ5, DQ4, DQ3, DQ2];

"Pass-thru byte enables latch
"Byte enables are latched on rising edge of PCI clock when PT_ADR is asserted
   pt_be_.ap     = reset;
   pt_be_.clk    = BPCLK;
   pt_be_.ce     = !PT_ADR_;
   pt_be_       := [PTBE3_,PTBE2_,PTBE1_,PTBE0_];

"=============================================================================
"   DSP-mapped Registers
"=============================================================================

"DSP-mapped register reads
   when          (EA_LO == DSP_CNTL_ADR)       then   {ED = dsp_cntl;}
      else when (EA_LO == DSP_STAT_ADR)        then   {ED = dsp_stat;}
      else when (EA_LO == DSP_DIPOPT_ADR)      then   {ED = dsp_dipopt;}
      else when (EA_LO == DSP_DIPBOOT_ADR)     then   {ED = dsp_dipboot;}
      else when (EA_LO == DSP_DSPOPT_ADR)      then   {ED = dsp_dspopt;}
      else when (EA_LO == DSP_DSPBOOT_ADR)     then   {ED = dsp_dspboot;}
      else when (EA_LO == DSP_FIFOSTAT_ADR)    then   {ED = dsp_fifostat;}
      else when (EA_LO == DSP_SDCNTL_ADR)      then   {ED = dsp_sdcntl;}
      else when (EA_LO == DSP_OSCB_ADR)        then   {ED = OSC_B_FREQ;}
      else when (EA_LO == DSP_SEM0_ADR)        then   {ED = dsp_sem0;}
      else when (EA_LO == DSP_SEM1_ADR)        then   {ED = dsp_sem1;}
      else                                            {ED = DSP_RSVD;}
```

```
"DSP-mapped register output enable
"Active when EMIF read to  CE1:0x01380000-0x0138FFFF range (CPLD regs)
"                          (CE1:0x01780000-0x0178FFFF for MAP1)
      ED.oe = CPLD_CS & !ARE_;

"-------------------------------------------------------------------------------
"DSP-mapped register writes

   "Control Register (CNTL)
   XCNTL1    := ED7;              "CNTL - bit 7
   XCNTL1.ar  = reset;            "default to 0
   XCNTL1.ce  = dsp_cntl_ce;
   XCNTL1.clk = AWE_;

   XCNTL0    := ED6;              "CNTL - bit 6
   XCNTL0.ar  = reset;            "default to 0
   XCNTL0.ce  = dsp_cntl_ce;
   XCNTL0.clk = AWE_;

   xreset    := ED5;              "CNTL - bit 5
   xreset.ar  = reset;            "default to no ext. reset
   xreset.ce  = dsp_cntl_ce;
   xreset.clk = AWE_;

   nmien     := ED4;              "CNTL - bit 4
   nmien.ar   = reset;            "default to NMI disabled
   nmien.ce   = dsp_cntl_ce;
   nmien.clk  = AWE_;

   nmisel    := ED3;              "CNTL - bit 3
   nmisel.ar  = reset;            "default to NMI from host
   nmisel.ce  = dsp_cntl_ce;
   nmisel.clk = AWE_;

   SP0SEL    := ED2;              "CNTL - bit 2
   SP0SEL.ar  = reset;            "default to McBSP0 -> DB
   SP0SEL.ce  = dsp_cntl_ce;
   SP0SEL.clk = AWE_;

   LED1_     := !ED1;             "CNTL - bit 1
   LED1_.ap   = reset;            "default to LED off
   LED1_.ce   = dsp_cntl_ce;
   LED1_.clk  = AWE_;

   LED0_     := !ED0;             "CNTL - bit 0
   LED0_.ap   = reset;            "default to LED off
   LED0_.ce   = dsp_cntl_ce;
   LED0_.clk  = AWE_;
```

```
   ”PCI FIFO Status Register (FIFOSTAT)
   pcimren   := ED1;                ”FIFOSTAT – bit 1
   pcimren.ar = reset;              ”default to PCI bus master read int disabled
   pcimren.ce = dsp_fifostat_ce;
   pcimren.clk= AWE_;

   pcimwen   := ED0;                ”FIFOSTAT – bit 0
   pcimwen.ar = reset;              ”default to PCI bus master write int disabled
   pcimwen.ce = dsp_fifostat_ce;
   pcimwen.clk= AWE_;

   ”SDRAM Control Register (SDCNTL)
   ce3sden   := ED1;                ”SDCNTL – bit 1
   ce3sden.ap = reset;              ”default to CE2 SDRAM enabled
   ce3sden.ce = dsp_sdcntl_ce;
   ce3sden.clk= AWE_;
   CE3_SDEN   = ce3sden & !reset;   ”disable SDRAM during reset for low power

   ce2sden   := ED0;                ”SDCNTL – bit 0
   ce2sden.ap = reset;              ”default to CE3 SDRAM enabled
   ce2sden.ce = dsp_sdcntl_ce;
   ce2sden.clk= AWE_;
   CE2_SDEN   = ce2sden & !reset;   ”disable SDRAM during reset for low power

   ”DSP Semaphore Registers (DSPSEMx)
   dspsem0    := ED0;               ”DSPSEM0 – bit 0
   dspsem0.ar  = reset # hostsem0;  ”DSP semaphore cleared at reset
   dspsem0.ce  = dsp_sem0_ce;
   dspsem0.clk = AWE_;

   dspsem1    := ED0;               ”DSPSEM1 – bit 0
   dspsem1.ar  = reset # hostsem1;  ”DSP semaphore cleared at reset
   dspsem1.ce  = dsp_sem1_ce;
   dspsem1.clk = AWE_;

”=============================================================================
”  PCI–mapped Registers
”=============================================================================

”PCI–mapped register reads

   when         (pt_addr == PCI_CNTL_ADR)     then {DQ = pci_cntl;}
   else when    (pt_addr == PCI_STAT_ADR)     then {DQ = pci_stat;}
   else when    (pt_addr == PCI_SWOPT_ADR)    then {DQ = pci_swopt;}
   else when    (pt_addr == PCI_SWBOOT_ADR)   then {DQ = pci_swboot;}
   else when    (pt_addr == PCI_DIPOPT_ADR)   then {DQ = pci_dipopt;}
   else when    (pt_addr == PCI_DIPBOOT_ADR)  then {DQ = pci_dipboot;}
   else when    (pt_addr == PCI_DSPOPT_ADR)   then {DQ = pci_dspopt;}
```

```
    else when    (pt_addr == PCI_DSPBOOT_ADR)  then {DQ = pci_dspboot;}
    else when    (pt_addr == PCI_CPLDREV_ADR)  then {DQ = pci_cpldrev;}
    else when    (pt_addr == PCI_SEM0_ADR)     then {DQ = pci_sem0;}
    else when    (pt_addr == PCI_SEM1_ADR)     then {DQ = pci_sem1;}
    else                                       {DQ = pci_reserve;}

"PCI add-on bus output enable when pcireg_oe set active
    DQ.oe = PCI.pcireg_oe;

"-------------------------------------------------------------------------------
"PCI-mapped register writes

    "EVM Control Register (CNTL)
    dspnmi      := DQ7;             "CNTL - bit 7
    dspnmi.ar    = reset;           "default to no NMI
    dspnmi.ce    = pci_cntl_ce;
    dspnmi.clk   = BPCLK;

    tbcinten    := DQ6;             "CNTL - bit 6
    tbcinten.ar  = reset;           "default to TBC interrupt disabled
    tbcinten.ce  = pci_cntl_ce;
    tbcinten.clk = BPCLK;

    hinten      := DQ5;             "CNTL - bit 5
    hinten.ar    = reset;           "default to DSP host interrupt disabled
    hinten.ce    = pci_cntl_ce;
    hinten.clk   = BPCLK;

    tbcrst      := DQ1;             "CNTL - bit 1
    tbcrst.ar    = reset;           "default to no TBC reset
    tbcrst.ce    = pci_cntl_ce;
    tbcrst.clk   = BPCLK;

    dsprst      := DQ0;             "CNTL - bit 0
    dsprst.ar    = reset;           "default to no DSP reset
    dsprst.ce    = pci_cntl_ce;
    dsprst.clk   = BPCLK;

    "Software Switch User Options Register (SW_OPT)
    h_clkmode   := DQ6;             "SWOPT - bit 6
    h_clkmode.ar = reset;           "default to x4 clock mode
    h_clkmode.ce = pci_swopt_ce;
    h_clkmode.clk= BPCLK;

    h_clksel    := DQ5;             "SWOPT - bit 5
    h_clksel.ar  = reset;           "default to OSC A-33.25 MHz
    h_clksel.ce  = pci_swopt_ce;
    h_clksel.clk = BPCLK;
```

```
h_endian    := DQ4;              ”SWOPT – bit 4
h_endian.ar = reset;             ”default to little endian
h_endian.ce = pci_swopt_ce;
h_endian.clk= BPCLK;

h_jtagsel   := DQ3;              ”SWOPT – bit 3
h_jtagsel.ar = reset;            ”default to external XDS510
h_jtagsel.ce = pci_swopt_ce;
h_jtagsel.clk= BPCLK;

h_user2     := DQ2;              ”SWOPT – bit 2
h_user2.ar   = reset;            ”default to USER2=0
h_user2.ce   = pci_swopt_ce;
h_user2.clk  = BPCLK;

h_user1     := DQ1;              ”SWOPT – bit 1
h_user1.ar   = reset;            ”default to USER1=0
h_user1.ce   = pci_swopt_ce;
h_user1.clk  = BPCLK;

h_user0     := DQ0;              ”SWOPT – bit 0
h_user0.ar   = reset;            ”default to USER0=0
h_user0.ce   = pci_swopt_ce;
h_user0.clk  = BPCLK;

”Software Switch Boot Mode Register (SWBOOT)
swsel       := DQ7;              ”SWBOOT – bit 7
swsel.ar     = reset;            ”default to DIP switch control
swsel.ce     = pci_swboot_ce;
swsel.clk    = BPCLK;

”SWBOOT – bits 4–0
”Default to Boot Mode 5 (No–Boot,Internal, MAP1)
h_bootmode    := [DQ4, DQ3, DQ2, DQ1, DQ0];
h_bootmode4.ar = reset;
h_bootmode3.ar = reset;
h_bootmode2.ap = reset;
h_bootmode1.ar = reset;
h_bootmode0.ap = reset;
h_bootmode.ce  = pci_swboot_ce;
h_bootmode.clk = BPCLK;

”Host Semaphore Registers (HOSTSEMx)
hostsem0     := DQ0;              ”HOSTSEM0 – bit 0
hostsem0.ar   = reset # dspsem0;  ”host semaphore cleared at reset
hostsem0.ce   = pci_sem0_ce;
hostsem0.clk  = BPCLK;
```

```
    hostsem1     := DQ0;              ”HOSTSEM1 – bit 0
    hostsem1.ar   = reset # dspsem1;  ”host semaphore cleared at reset
    hostsem1.ce   = pci_sem1_ce;
    hostsem1.clk  = BPCLK;


”=============================================================================
”   Float Output Control (for testing)
”=============================================================================

    ”Outputs enabled when FLOAT_ is high (pull–up on board)
    ”Useful for in–circuit testing.

    DSP2AOD_.OE     = FLOAT_;
    DSP2PBD_.OE     = FLOAT_;
    DSP2XD_.OE      = FLOAT_;
    TBCA0_HBE0_.OE  = FLOAT_;
    TBCA1_HBE1_.OE  = FLOAT_;
    TBCA2_HHWIL.OE  = FLOAT_;
    TBCA3_HCNTL0.OE = FLOAT_;
    TBCA4_HCNTL1.OE = FLOAT_;
    HCS_.OE         = FLOAT_;
    HDS1_.OE        = FLOAT_;
    HRW.OE          = FLOAT_;
    TBC_WR_.OE      = FLOAT_;
    TBC_RD_.OE      = FLOAT_;
    TBC_RST_.OE     = FLOAT_;
    PCI_FLT_.OE     = FLOAT_;
    PT_ADR_.OE      = FLOAT_;
    PT_RDY_.OE      = FLOAT_;
    AO_SEL_.OE      = FLOAT_;
    AO_WR_.OE       = FLOAT_;
    AO_RD_.OE       = FLOAT_;
    AO_ADR6.OE      = FLOAT_;
    AO_ADR5.OE      = FLOAT_;
    AO_ADR4.OE      = FLOAT_;

    AO_ADR3.OE      = FLOAT_;
    AO_ADR2.OE      = FLOAT_;
    AO_BE3_.OE      = FLOAT_;
    AO_BE2_.OE      = FLOAT_;
    AO_BE1_.OE      = FLOAT_;
    AO_BE0_.OE      = FLOAT_;
    WRFIFO_.OE      = FLOAT_;
    RDFIFO_.OE      = FLOAT_;
    PCIMRD_INT.OE   = FLOAT_;
```

```
        PCIMWR_INT.OE   = FLOAT_;
        PC_INT.OE       = FLOAT_;
        BOOTMODE0.OE    = FLOAT_;
        BOOTMODE1.OE    = FLOAT_;
        BOOTMODE2.OE    = FLOAT_;
        BOOTMODE3.OE    = FLOAT_;
        BOOTMODE4.OE    = FLOAT_;
        CLKMODE.OE      = FLOAT_;
        LENDIAN.OE      = FLOAT_;
        OSC_A_EN_.OE    = FLOAT_;
        OSC_B_EN_.OE    = FLOAT_;
        JTAGSEL.OE      = FLOAT_;
        NMI.OE          = FLOAT_;
        PCI_INT.OE      = FLOAT_;
        DSP_RST_.OE     = FLOAT_;
        ARDY.OE         = FLOAT_;
        LED1_.OE        = FLOAT_;
        LED0_.OE        = FLOAT_;
        SP0SEL.OE       = FLOAT_;
        XCNTL1.OE       = FLOAT_;
        XCNTL0.OE       = FLOAT_;
        CODEC_CS_.OE    = FLOAT_;
        CODEC_WR_.OE    = FLOAT_;
        CODEC_RD_.OE    = FLOAT_;
        CE3_SDEN.OE     = FLOAT_;
        CE2_SDEN.OE     = FLOAT_;
        EXT_RST_.OE     = FLOAT_;
        MAN_RST_.OE     = FLOAT_;

END
```

```
MODULE EVMRESET

interface (BRD_RST_,SW_RST_,VCC2BAD_,PCI_RST_,PCI_DET_,xreset,dsprst,tbcrst ->
          MAN_RST_,DSP_RST_,TBC_RST_,EXT_RST_);

TITLE 'TMS320C62x EVM Reset Logic'

"DWG NAME  TMS320C6x Evaluation Module (EVM)
"ASSY#     D600830-0001
"PAL #     U12, 830*
"PAL TYPE  Altera EPM7256SQC208-10 (256-Macrocells, 10-ns, 208-PQFP)
"COMPANY   Texas Instruments Incorporated / DNA Enterprises, Inc.
"ENGR      Brian G. Carlson
"DATE      02/21/98
"TOOLS     Synario ABEL 6.5 / Altera Max Plus II 8.2

"DESCRIPTION
       "This module controls the various types of reset signals that
       "are used on the 'C6x EVM board.  Signals are generated that
       "reset the board, DSP, JTAG TBC and external daughterboard
       "independently.  All resets are asserted upon 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.

"INPUTS
       BRD_RST_    pin;    "Reset from voltage supervisor
       SW_RST_     pin;    "Reset from push button switch
       VCC2BAD_    pin;    "VCC bad from the voltage supervisor
       PCI_RST_    pin;    "Reset from PCI controller
       PCI_DET_    pin;    "PCI detection (when 0)
       xreset      pin;    "Ext. reset from DSP-mapped register
       dsprst      pin;    "DSP reset from PCI-mapped register
       tbcrst      pin;    "TBC reset from PCI-mapped register

"OUTPUTS
       MAN_RST_    pin;    "Manual reset to voltage supervisor
       DSP_RST_    pin;    "DSP reset
       TBC_RST_    pin;    "TBC reset
       EXT_RST_    pin;    "Daughter board reset

"Contants

        h,l,x = 1, 0, .X. ;
```

```
Equations

   "--------------------------------------------------------------------
   "Reset Control

      "Manual Reset – Asserted when PCI or switch reset.
      "NOTE: SW_RST_ will bounce during reset pushbutton use, but
      "      the MAX708 voltage supervisor effectively provides a
      "      debounce by holding the reset line active for
      "      140–200 milliseconds after the last low value.
      !MAN_RST_  = (!PCI_RST_ & !PCI_DET_) # !SW_RST_;

      "DSP Reset – Asserted when DSP software reset is asserted,
      "            DSP core voltage bad or board reset.
      "            Allows host to control DSP reset for HPI booting.
      !DSP_RST_  = dsprst # !VCC2BAD_ # !BRD_RST_;

      "TBC Reset – Asserted upon host assertion or board reset.
      !TBC_RST_  = tbcrst # !BRD_RST_;

      "External (Daughter Board) Reset – Asserted upon  DSP firmware
      "           assertion or board reset.
      !EXT_RST_  = xreset # !BRD_RST_;

END
```

```
MODULE EVMSWMUX
interface (h_clkmode,h_clksel,h_endian,h_jtagsel,h_user2..h_user0,
          swsel,h_bootmode4..h_bootmode0,S_CLKMODE,S_CLKSEL,
          S_ENDIAN,S_JTAGSEL,S_USER2..S_USER0,
          S_BOOTMODE4..S_BOOTMODE0 ->
          BOOTMODE4..BOOTMODE0,CLKMODE,LENDIAN,USER2..USER0,
          clksel,JTAGSEL);
TITLE 'TMS320C6x EVM User Options (Switch) Mux'
"DWG NAME  TMS320C6x Evaluation Module (EVM)
"ASSY#     D600830-0001
"PAL #     U12, 830*
"PAL TYPE  Altera EPM7256SQC208-10 (256-Macrocells, 10-ns, 208-PQFP)
"COMPANY   Texas Instruments Incorporated / DNA Enterprises, Inc.
"ENGR      Brian G. Carlson
"DATE      02/21/98
"TOOLS     Synario ABEL 6.5 / Altera Max Plus II 8.2
"DESCRIPTION
   "This module provides multiplexing between hardware DIP switch
   "user options and software switches that are controlled by the
   "host software.  The CPLD defaults to the use of the DIP switch
   "settings at power-up and board reset.  The multiplexer is
   "controlled by the host software using the PCI memory-mapped
   "CPLD SWBOOT register's SWSEL bit.  When SWSEL is 0, the DIP
   "switches are used, and when SWSEL is 1, the software switches
   "are used.
"INPUTS
   "-------------------------------------------------------------------------------
   "From Host (Software Switch) Inputs
      h_clkmode   pin;    "Host clock mode
      h_clksel    pin;    "Host clock select
      h_endian    pin;    "Host endian select
      h_jtagsel   pin;    "Host JTAG select
      h_user2     pin;    "Host user-defined switches 2..0
      h_user1     pin;
      h_user0     pin;
      swsel       pin;    "DIP / software switch select
      h_bootmode4 pin;    "Host DSP boot mode selection
      h_bootmode3 pin;
      h_bootmode2 pin;
      h_bootmode1 pin;
      h_bootmode0 pin;
      h_bootmode = [h_bootmode4..h_bootmode0];
```

```
    "-------------------------------------------------------------------------------
    "From the DIP switch
       S_CLKMODE   pin;    "Selects x1 (No PLL) or x4 (PLL) Clock Mode
       S_CLKSEL    pin;    "Selects output of oscillator A or oscillator B
       S_ENDIAN    pin;    "Selects big or little endian memory addressing
       S_JTAGSEL   pin;    "Selects internal or external JTAG emulation
       S_USER2     pin;    "User-defined switches 2-0
       S_USER1     pin;
       S_USER0     pin;
       S_BOOTMODE4 pin;    "Selects No-boot, HPI-boot or ROM-boot
       S_BOOTMODE3 pin;
       S_BOOTMODE2 pin;
       S_BOOTMODE1 pin;
       S_BOOTMODE0 pin;
       S_BOOTMODE = [S_BOOTMODE4..S_BOOTMODE0];

"OUTPUTS
    "-------------------------------------------------------------------------------
    " To TMS320C6201 DSP
       BOOTMODE4   pin;     "DSP boot mode [4..0]
       BOOTMODE3   pin;
       BOOTMODE2   pin;
       BOOTMODE1   pin;
       BOOTMODE0   pin;
       BOOTMODE = [BOOTMODE4..BOOTMODE0];

       CLKMODE     pin;    "DSP clock mode
       LENDIAN     pin;    "DSP little endian select
    "-------------------------------------------------------------------------------
    " To Internal CPLD Register
       USER2       pin;    "User-defined options [2..0]
       USER1       pin;
       USER0       pin;

    "-------------------------------------------------------------------------------

    " To Internal CPLD Clock Selection Logic
       clksel      pin;    "DSP clock source selection
```

```
   ”-------------------------------------------------------------------------------
   ” To JTAG Multiplexer
      JTAGSEL     pin;    ”JTAG selection (ext/int)

”Constants

   h,l,x = 1, 0, .X. ;

Equations

      ”Multiplexer selects host software switches when swsel=1
      ”Note: h_clkmode and h_endian selections are inverted from
      ”      host selections to match polarity of DSP inputs.

      when (swsel) then

         {
         CLKMODE = !h_clkmode;
         clksel  = h_clksel;
         LENDIAN = !h_endian;
         JTAGSEL = h_jtagsel;
         USER2   = h_user2;
         USER1   = h_user1;
         USER0   = h_user0;
         BOOTMODE= h_bootmode;
         }

      ”Multiplexer selects DIP switches when swsel=0
      ”Note: S_CLKMODE and H_ENDIAN selections are inverted from
      ”      switch selections to match polarity of DSP inputs.
      else
         {
         CLKMODE = !S_CLKMODE;
         clksel  = S_CLKSEL;
         LENDIAN = !S_ENDIAN;
         JTAGSEL = S_JTAGSEL;
         USER2   = S_USER2;
         USER1   = S_USER1;
         USER0   = S_USER0;
         BOOTMODE= S_BOOTMODE;
         }

END
```

```
MODULE EVMCKSEL

interface (CLKA_DIV2,clksel -> osc_a_, osc_b_);

TITLE 'TMS320C6x EVM Clock Select Logic'

"DWG NAME  TMS320C6x Evaluation Module (EVM)
"ASSY#     D600830-0001
"PAL #     U12, 830*
"PAL TYPE  Altera EPM7256SQC208-10 (256-Macrocells, 10-ns, 208-PQFP)
"COMPANY   Texas Instruments Incorporated / DNA Enterprises, Inc.
"ENGR      Brian G. Carlson
"DATE      02/21/98
"TOOLS     Synario ABEL 6.5 / Altera Max Plus II 8.2

"DESCRIPTION

   "This module implements the DSP source clock selection.  The DSP
   "source clock (CLKIN) can either be 33.25 or 40 MHz.  The two oscillator
   "outputs are buffered using a SN74LVT125 quad buffer device which
   "has independent output enable controls.  The two buffered clock outputs
   "are tied together.  The state machine in this module ensures that only
   "one clock output is enabled at any one time by implementing a
   "break-before-make switch.  The source clock selection can be made
   "using a DIP switch (SW2-7) or a software switch (DSPBOOT-SWSEL).

"INPUTS

      CLKA_DIV2   pin;    "Free running clk (16.625 MHz) used for clk select state machine
      clksel      pin;    "Clock select at output of switch mux

"OUTPUTS

      osc_a_      pin;     "Oscillator A buffer output enable (active low)
      osc_b_      pin;     "Oscillator B buffer output enable (active low)

"Internal Registers

      clksel_db1  node istype 'reg_d, dc, buffer';  "delayed clock select
      clksel_db2  node istype 'reg_d, dc, buffer';  "delayed clock select

"Constants

      h,l,x = 1, 0, .X. ;
```

```
Equations
   "--------------------------------------------------------------------
   "NOTE: The selected clock must be enabled during reset
   "      so that the DSP's PLL locks before reset is released.
   "      This means that the FF's should be free-running without
   "      being held by reset.
   "--------------------------------------------------------------------

   "clksel = 0 selects oscillator A (33.25 MHz)
   "clksel = 1 selects oscillator B (40.00 MHz)

   "Switch mux clksel output fed to cascased D flip-flops to determine
   "clock selection transition.  This creates a break-before-make clock
   "switch that prevents contention between the two clock buffer outputs.

   clksel_db1.d   = clksel;
   clksel_db1.clk = CLKA_DIV2;

   clksel_db2    := clksel_db1;
   clksel_db2.clk = CLKA_DIV2;

   "Oscillator A selected only when clksel low for two consecutive clocks
   "This signal controls the SN74LVT125 buffer output enable for OSC A.
   osc_a_ =  clksel_db1 #  clksel_db2;

   "Oscillator B selected only when clksel high for two consecutive clocks
   "This signal controls the SN74LVT125 buffer output enable for OSC B.
   osc_b_ = !clksel_db1 # !clksel_db2;

END
```

```
MODULE EVMDECOD

interface (RESET,PCI_DET_,PCICLK,DSPCLK,
          CE1_,CE2_,CE3_,ARE_,AWE_,EA21..EA16,
          ao_bsy,emif_ack,XRDY,ce2sden,ce3sden,CLKMODE –>
          DSP2AOD_,DSP2PBD_,DSP2XD_,are_r_,awe_r_,emif_req,
          CPLD_CS,CODEC_CS_,CODEC_WR_,CODEC_RD_,ARDY);

TITLE 'TMS320C6x EVM Memory Decode Logic'

"DWG NAME  TMS320C6x Evaluation Module (EVM)
"ASSY#     D600830–0001
"PAL #     U12, 830*
"PAL TYPE  Altera EPM7256SQC208–10 (256–Macrocells, 10–ns, 208–PQFP)
"COMPANY   Texas Instruments Incorporated / DNA Enterprises, Inc.
"ENGR      Brian G. Carlson
"DATE      02/21/98
"TOOLS     Synario ABEL 6.5 / Altera Max Plus II 8.2

"DESCRIPTION

   "This module provides DSP EMIF memory decoding to control
   "asynchronous memory accesses to the CE1 memory space.   The devices
   "in the CE1 memory space include CPLD registers, the PCI controller,
   "the audio codec and the expansion memory (daughterboard).  Decode
   "logic controls data bus transceivers that connect the DSP's EMIF
   "data bus to the peripheral, PCI add–on and external daughterboard
   "data busses.  The asynchronous ARE_ and AWE_ EMIF signals are
   "synchronized for use by the PCI controller state machine.  Asynchronous
   "memory access ready (RDY) generation is implemented for each type
   "of device access in the CE1 space.  Audio codec chip select and
   "read/write strobes are controlled with a state machine to ensure
   "proper single–cycle and back–to–back access timing.

"REVISIONS
"   03/05/98 – Modified DSP EMIF ready logic to provide ARDY for invalid
"              async accesses to non–CE1 memory spaces to prevent the
"              EMIF from locking–up at power–up when invalid code is
"              executed in the no–boot mode.  Also, extended codec_rdy
"              flag assertion to maintain ARDY assertion until ARE– or
"              AWE– is deasserted to support any strobe period >= 3
"              CLKOUT1 clocks.

"INPUTS

   "–------------------------------------------------------------------------------
   " From Voltage Supervisor
      RESET       pin;    "Active high board reset
```

```
    "-------------------------------------------------------------------------------
    " From S5933 PCI Controller
       PCI_DET_    pin;    "PCI detection indicator (0=PCI,1=Standalone)
       PCICLK      pin;    "Buffered PCI clock (33 MHz max)

    "-------------------------------------------------------------------------------
    " From TMS320C6201 DSP
       DSPCLK      pin;    "DSP's CLKOUT2
       CE1_        pin;    "EMIF CE1 memory space enable
       CE2_        pin;    "EMIF CE2 memory space enable
       CE3_        pin;    "EMIF CE3 memory space enable
       ARE_        pin;    "EMIF asynchronous memory read strobe
       AWE_        pin;    "EMIF asynchronous memory write strobe

       "EMIF word address bits [21..16]
       EA21        pin;
       EA20        pin;
       EA19        pin;
       EA18        pin;
       EA17        pin;
       EA16        pin;
       EA   = [EA21, EA20, EA19, EA18, EA17, EA16];

    "-------------------------------------------------------------------------------
    " From Add-on State Machine
       ao_bsy      pin;    "Add-on state machine busy flag
       emif_ack    pin;    "Add-on state machine EMIF access acknowledge

    "-------------------------------------------------------------------------------
    " From Daughter Board
       XRDY        pin;    "EMIF asynchronous memory access ready from DB

    "-------------------------------------------------------------------------------
    " From DSP-mapped Registers
       ce2sden     pin;    "CE2 SDRAM enable
       ce3sden     pin;    "CE3 SDRAM enable

    "-------------------------------------------------------------------------------
    " From Internal Switch Multiplexer
       CLKMODE     pin;    "Clock mode

"OUTPUTS

    "-------------------------------------------------------------------------------
    " To Data Buffer
       DSP2AOD_    pin;    "Enables ED[31..0] to AOD[31..0]
       DSP2PBD_    pin;    "Enables ED[7..0] to PBD[7..0]
       DSP2XD_     pin;    "Enables ED[31..0] to XD[31..0]
```

```
   "-------------------------------------------------------------------------------
   " To Add-on State Machine
      are_r_      pin istype 'reg_d, dc, buffer';  "Synchronized ARE_
      awe_r_      pin istype 'reg_d, dc, buffer';  "Synchronized AWE_
      emif_req    pin istype 'reg_d, dc, buffer';  "Synchronized EMIF req.

   "-------------------------------------------------------------------------------
   " To CPLD Registers
      CPLD_CS     pin istype 'com';    "CPLD DSP register chip select

   "-------------------------------------------------------------------------------
   " To CS4231 Audio Codec
      CODEC_CS_   pin istype 'com';    "Audio codec chip select
      CODEC_WR_   pin istype 'com';    "Audio codec write strobe
      CODEC_RD_   pin istype 'com';    "Audio codec read strobe

   "-------------------------------------------------------------------------------
   " To TMS320C6201 DSP
      ARDY        pin istype 'com';    "EMIF async memory access ready

"Internal Registers
      dspclk2     node istype 'reg_d, dc, buffer';  "CLKOUT2/2
      codec_rdy   node istype 'reg_d, dc, buffer';  "Audio codec ready flag
      codec_rd_r  node istype 'reg_d, dc, buffer';  "Audio codec read reg
      codec_wr_r  node istype 'reg_d, dc, buffer';  "Audio codec write reg

   "Audio codec read/write strobe generation state machine (acr)
      acr4, acr3, acr2, acr1, acr0  node istype 'reg_d, dc, buffer';
      acr = [acr4..acr0];

"Constants

   h,l,x = 1, 0, .X. ;

   "EMIF CE1_ Addresses [EA21..EA16]              MAP0         MAP1
      PCI_REG_ADDR  = [1, 1, 0, 0, 0, 0];    " 0x0130xxxx / 0x0170xxxx
      PCI_FIFO_ADDR = [1, 1, 0, 0, 0, 1];    " 0x0131xxxx / 0x0171xxxx
      CODEC_REG_ADDR= [1, 1, 0, 0, 1, 0];    " 0x0132xxxx / 0x0172xxxx
      DSP_REG_ADDR  = [1, 1, 1, 0, 0, 0];    " 0x0138xxxx / 0x0178xxxx

   "Audio codec read/write strobe generation state machine states
   "  State Name   State Value (Bin)         (Hex)
      ACS_IDLE   = [ 0, 0, 0, 0, 0 ];        "0x00
      ACS_BEGIN  = [ 0, 0, 0, 0, 1 ];        "0x01
      ACS_DELAY1 = [ 0, 0, 0, 1, 0 ];        "0x02
      ACS_DELAY2 = [ 0, 0, 1, 0, 0 ];        "0x04
      ACS_READ1  = [ 0, 1, 0, 0, 0 ];        "0x08
      ACS_READ2  = [ 0, 1, 0, 0, 1 ];        "0x09
```

```
        ACS_READ3  = [ 0, 1, 0, 1, 0 ];        "0x0A
        ACS_READ4  = [ 0, 1, 0, 1, 1 ];        "0x0B
        ACS_READ5  = [ 0, 1, 1, 0, 0 ];        "0x0C
        ACS_WRITE1 = [ 1, 0, 0, 0, 0 ];        "0x10
        ACS_WRITE2 = [ 1, 0, 0, 0, 1 ];        "0x11
        ACS_WRITE3 = [ 1, 0, 0, 1, 0 ];        "0x12
        ACS_WRITE4 = [ 1, 0, 1, 0, 0 ];        "0x14
        ACS_WRITE5 = [ 1, 1, 0, 0, 0 ];        "0x18
        ACS_WRITE6 = [ 1, 1, 0, 0, 1 ];        "0x19
        ACS_WRITE7 = [ 1, 1, 0, 1, 0 ];        "0x1A

    Equations

        "-------------------------------------------------------------------------------
        "DSP to Add-on, Peripheral and Expansion Bus Data Buffers Control
        "-------------------------------------------------------------------------------
            "Connect ED[31..0] to AOD[31..0]
            "ED[7..0] is connected to AOD[7..0] only when DSP2PB_ is low also.
            !DSP2AOD_ =  ((EA == PCI_REG_ADDR) # (EA == PCI_FIFO_ADDR)) &
                         !CE1_ & !ao_bsy & (!ARE_ # !AWE_);

            "Connect ED[7..0] to PBD[7..0]
            "Also connects ED[7..0] to AOD[7..0] when DSP2AOD_ is low.
            !DSP2PBD_ = !CE1_ & EA21 & EA20 & (!ARE_ # !AWE_);

            "Connects ED[31..0] to XD[31..0] (Daughter Board)
            !DSP2XD_  =  ((!CE1_ & !EA21)    #
                         (!CE1_ & !EA20)    #  "EMIF CE1_ address = $xx00xxxx-xx2xxxxx
                         (!CE2_ & !ce2sden) #  "EMIF CE2_ address & SDRAM disabled
                         (!CE3_ & !ce3sden));  "EMIF CE3_ address & SDRAM disabled

        "-------------------------------------------------------------------------------
        "Async Read/Write Strobe Synchronization
        "-------------------------------------------------------------------------------
        "A synchronizer is required to elimination metastability
        "problems that can be caused by sampling these asynchronous signals
        "when they transition within the setup period.  The EMIF asynchronous
        "ARE_ and AWE_ signals are synchronized to the PCI clock.

            "Add-on state machine ARE_ synchronization
            are_r_.ap   = RESET;
            are_r_.clk  = PCICLK;
            are_r_     := ARE_;

            "Add-on state machine AWE_ synchronization
            awe_r_.ap   = RESET;
            awe_r_.clk  = PCICLK;
            awe_r_     := AWE_;
```

```
"-------------------------------------------------------------------------------
"EMIF PCI Controller Access Request Synchronization
"-------------------------------------------------------------------------------
"Synchronize EMIF request to add-on bus (PCI register or FIFO access)
"A dual-FF synchronizer is required to elimination metastability
"problems that can be caused by sampling these asynchronous signals
"when they transition within the setup period."

   emif_req    :=   !CE1_ &
                    ((EA == PCI_REG_ADDR) # (EA == PCI_FIFO_ADDR));
   emif_req.ar  =   RESET;
   emif_req.clk =   PCICLK;

"-------------------------------------------------------------------------------
"CPLD Register Chip Select
"-------------------------------------------------------------------------------
   CPLD_CS    = !CE1_ & (EA == DSP_REG_ADDR);

"-------------------------------------------------------------------------------
"DSP Asynchronous Memory Access ARDY Generation
"-------------------------------------------------------------------------------
   "PCI controller register or FIFO access
   "PCI controller accesses are ready when the PCI state machine
   "acknowledges the access.
   "NOTE: If not in PCI slot, then PCI accesses do not lock up
   "      the EMIF waiting for a response since ARDY is asserted
   "      when PCI_DET_ is high.
   when (!CE1_ & ((EA == PCI_REG_ADDR) # (EA == PCI_FIFO_ADDR))) then {
         ARDY = (emif_ack & (!ARE_ # !AWE_)) # PCI_DET_;
   }
   else

   "CPLD DSP register accesses are ready when ARE_ or AWE_ goes low
   when (!CE1_ & (EA == DSP_REG_ADDR)) then {
         ARDY = !ARE_ # !AWE_;
   }
   else

   "Audio codec register accesses are ready when codec_rdy set
   when (!CE1_ & (EA == CODEC_REG_ADDR)) then  {
         ARDY = codec_rdy;
   }
   else
```

```
    ”External memory access in CE1_, CE2_ or CE3_ uses XRDY from DB
    when  ((!CE1_ & !EA21)     #
          (!CE1_ & !EA20)     #  ”EMIF CE1_ address = $xx00xxxx–xx2xxxxx
          (!CE2_ & !ce2sden)  #  ”EMIF CE2_ address & SDRAM disabled
          (!CE3_ & !ce3sden))    ”EMIF CE3_ address & SDRAM disable
    then  {
          ARDY = XRDY;
    }

    ”NOTE: ARDY is defaulted to asserted for all other CE1
    ”      accesses to prevent the DSP from being locked up
    ”      on an invalid memory access.
    ”BGC–03/05/98–Rev. 1– Added non–CE1 async ARDY generation to prevent
    ”             EMIF bus lock on invalid memory access during power–up
    ”             when no valid progam is in memory.
    else ARDY = !CE1_ # (CE1_ & !ARE_) # (CE1_ & !AWE_);

”––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––
”Generate DSPCLK2/2 for synchronization (33.25 / 40 MHz)
”––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––
dspclk2    := !dspclk2;
dspclk2.clk = DSPCLK;

”––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––
”CS4231A Audio Codec Interface Support
”––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––
   ”Audio codec chip select
   !CODEC_CS_ = (EA == CODEC_REG_ADDR) & !CE1_;

   ”Audio codec read/write strobes are internal register outputs
   CODEC_RD_  = codec_rd_r;
   CODEC_WR_  = codec_wr_r;

   ”Internal read/write strobe registers
   codec_rd_r.ap  = RESET # ARE_;    ”default RD_ to inactive at reset
   codec_rd_r.clk = dspclk2;
   codec_wr_r.ap  = RESET;           ”default WR_ to inactive at reset
   codec_wr_r.clk = dspclk2;

   ”Internal audio codec ready register
   codec_rdy.ar   = RESET;           ”default to not ready
   codec_rdy.clk  = dspclk2;

   ”Audio codec state machine for RD_/WR_ generation
   acr.ar  = RESET # (ARE_ & AWE_);  ”default to idle state ACS_IDLE at reset
   acr.clk = dspclk2;
```

```
state_diagram acr
   "-------------------------------------------------
   " Wait for audio codec access
   "-------------------------------------------------
   state ACS_IDLE:

      "Audio codec access not ready
      codec_rdy  := 0;

      "Read/write strobes are inactive during idle state
      codec_rd_r := 1;
      codec_wr_r := 1;

      "Wait for audio codec access
      "Note: State machine reset will hold in this state
      "      until ARE_ or AWE_ is low
      "If CLKMODE is x4 mode, then all states are required to
      "ensure proper back-to-back cycle delay
      if (!CODEC_CS_ & !CLKMODE) then ACS_BEGIN;
      else

      "if CLKMODE is x1 and read, then skip to ACS_READ3
      "since CLKOUT1 period is long enough to ensure proper
      "back-to-back cycle delay
      if (!CODEC_CS_ & !CLKMODE & !ARE_) then ACS_READ3;
      else

      "if CLKMODE is x1 and write, then skip to ACS_WRITE5
      "since CLKOUT1 period is long enough to ensure proper
      "back-to-back cycle delay
      if (!CODEC_CS_ & !CLKMODE & !AWE_) then ACS_WRITE5;
      else ACS_IDLE;

      "-------------------------------------------------
      " Audio codec access is in progress
      "-------------------------------------------------
      " Delay is required between back-to-back codec
      " accesses, so this it is enforced at the start
      " of each cycle.

      state ACS_BEGIN:
         "Audio codec access not ready
         codec_rdy  := 0;

         "Read/write strobes are inactive
         codec_rd_r := 1;
         codec_wr_r := 1;
         goto ACS_DELAY1;
```

```
   state ACS_DELAY1:
      "Audio codec access not ready
      codec_rdy  := 0;

      "Read/write strobes are still inactive
      codec_rd_r := 1;
      codec_wr_r := 1;
      goto ACS_DELAY2;

   state ACS_DELAY2:
      "Audio codec access not ready
      codec_rdy  := 0;

      "Read/write strobes are still inactive
      codec_rd_r := 1;
      codec_wr_r := 1;

      "Now determine if read or write
      if (!ARE_) then ACS_READ1;
      else
      if (!AWE_) then ACS_WRITE1;
      else            ACS_IDLE;

   "-------------------------------------------------
   " Audio codec read access
   "-------------------------------------------------
   state ACS_READ1:
   "Audio codec access not ready
   codec_rdy  := 0;

   "Assert read strobe to codec (write inactive)
   codec_rd_r := 0;
   codec_wr_r := 1;
   goto ACS_READ2;

state ACS_READ2:
   "Audio codec access not ready
   codec_rdy  := 0;

   "Read strobe still asserted
   codec_rd_r := 0;
   codec_wr_r := 1;
   goto ACS_READ3;

state ACS_READ3:
   "Audio codec access not ready
   codec_rdy  := 0;
```

```
   ”Read strobe still asserted
   codec_rd_r := 0;
   codec_wr_r := 1;
   goto ACS_READ4;

state ACS_READ4:
   ”Audio codec access ready
   codec_rdy  := 1;

   ”Read strobe still asserted
   codec_rd_r := 0;
   codec_wr_r := 1;
   goto ACS_READ5;

state ACS_READ5:
   ”Audio codec access ready
   ”BGC–03/05/98–Rev. 1–Changed codec_rdy assignment to 1
   ”                   to support unknown strobe period.
   ”                   This prevents the EMIF bus from
   ”                   locking up waiting for ARDY.
   codec_rdy  := 1;

   ”Read strobe still asserted
   codec_rd_r := 0;
   codec_wr_r := 1;

   ”Loop until ARE_ goes high
   if ARE_ then ACS_IDLE with {
      codec_rd_r := 1;
      codec_wr_r := 1;
   ”BGC–03/05/98–Rev. 1–Added codec_rdy assignment to
   ”                   clear codec_rdy flag.
      codec_rdy := 0;
   }
   else ACS_READ5;

”-------------------------------------------------
” Audio codec write access
”-------------------------------------------------
state ACS_WRITE1:
   ”Audio codec access not ready
   codec_rdy  := 0;

   ”Assert write strobe to codec (read inactive)
   codec_rd_r := 1;
   codec_wr_r := 0;
   goto ACS_WRITE2;
```

```
state ACS_WRITE2:
   ”Audio codec access not ready
   codec_rdy  := 0;

   ”Write strobe still asserted
   codec_rd_r := 1;
   codec_wr_r := 0;
   goto ACS_WRITE3;

state ACS_WRITE3:
   ”Audio codec access not ready
   codec_rdy  := 0;

   ”Write strobe still asserted
   codec_rd_r := 1;
   codec_wr_r := 0;
   goto ACS_WRITE4;

state ACS_WRITE4:
   ”Audio codec access not ready
   codec_rdy  := 0;

   ”Write strobe still asserted
   codec_rd_r := 1;
   codec_wr_r := 0;
   goto ACS_WRITE5;

state ACS_WRITE5:
   ”Audio codec access not ready
   codec_rdy  := 0;

   ”Write strobe still asserted
   codec_rd_r := 1;
   codec_wr_r := 0;
   goto ACS_WRITE6;

state ACS_WRITE6:
   ”Audio codec access ready
   codec_rdy  := 1;

   ”Deassert write strobe to latch data into codec
   codec_rd_r := 1;
   codec_wr_r := 1;
   goto ACS_WRITE7;
```

```
            state ACS_WRITE7:
               ”Audio codec access ready
               ”BGC–03/05/98–Rev. 1–Changed codec_rdy assignment to 1
               ”                   to support unknown strobe period.
               ”                   This prevents the EMIF bus from
               ”                   locking up waiting for ARDY.
               codec_rdy  := 1;

               ”Read and write strobes both high
               codec_rd_r := 1;
               codec_wr_r := 1;

               ”Loop until AWE_ goes high
               if AWE_ then ACS_IDLE with  {
               ”BGC–03/05/98–Rev. 1–Added codec_rdy assignment to
               ”                   clear codec_rdy flag.
                  codec_rdy := 0;
               }
               else ACS_WRITE7;

END
```

```
MODULE EVMINT

interface (RESET,DSP_HINT_,TBC_INT_,PCICLK,PCI_DET_,PCI_IRQ_,
          RDEMPTY,WRFULL,CODEC_IRQ, hinten, tbcinten,
          dspnmi,nmisel,nmien,pcimren,pcimwen,RDFIFO_,WRFIFO_ ->
          PC_INT,NMI,PCI_INT,PCIMRD_INT,PCIMWR_INT);

TITLE 'TMS320C6x EVM Interrupt Control'

"DWG NAME  TMS320C6x Evaluation Module (EVM)
"ASSY#     D600830-0001
"PAL #     U12, 830*
"PAL TYPE  Altera EPM7256SQC208-10 (256-Macrocells, 10-ns, 208-PQFP)
"COMPANY   Texas Instruments Incorporated / DNA Enterprises, Inc.
"ENGR      Brian G. Carlson
"DATE      02/21/98
"TOOLS     Synario ABEL 6.5 / Altera Max Plus II 8.2

"DESCRIPTION
   "This module controls interrupts to the host (via the PCI controller) and the 'C6201 DSP.
   "The DSP host interrupt and the JTAG TBC interrupt can be used to interrupt the host.
   "This module implements falling edge detectors to determine when these source interrupts
   "occur to force a PCI mailbox interrupt in the S5933 PCI controller which causes an
   "INTA# host interrupt.  Logic is included to implement a DSP NMI interrupt source
   "multiplexer that selects between the host and codec-controlled NMI interrupts.  The
   "PCI controller add-on IRQ# interrupt is passed to the DSP via its EXT_INT4 input.
   "Additionally, PCI bus master read (EXT_INT5) and (EXT_INT6) interrupts are generated
   "by state machines implemented in this module that monitor the PCI controller's FIFO
   "RDEMPTY and WRFULL flags to determine when data can be transferred between the
   "PCI controller and the DSP memory space.  These interrupts can be used by the DSP to
   "trigger an interrupt service routine, or more typically, be used to trigger background DMA
   "data transfers.  Optionally, DSP software could even poll the state of these interrupts, or the
   "FIFO flags themselves, to control the data transfers.

"INPUTS

   "-------------------------------------------------------------------------------
   " From Voltage Supervisor
      RESET       pin;    "Active high board reset

   "-------------------------------------------------------------------------------
   " From TMS320C6201 DSP
      DSP_HINT_   pin;    "DSP HPI host interrupt

   "-------------------------------------------------------------------------------
   " From JTAG TBC
      TBC_INT_    pin;    "TBC host interrupt
```

```
   "-------------------------------------------------------------------------------
   " From PCI Controller
      PCICLK      pin;    "Buffered PCI clock (33 MHz max)
      PCI_DET_    pin;    "PCI detection indicator (0=PCI,1=Standalone)
      PCI_IRQ_    pin;    "Interrupt to add-on device
      RDEMPTY     pin;    "Read FIFO empty flag
      WRFULL      pin;    "Write FIFO full flag
   "-------------------------------------------------------------------------------
   " From CS4231A Audio Codec
      CODEC_IRQ   pin;    "Audio codec interrupt
   "-------------------------------------------------------------------------------
   " From PCI-mapped Registers
      hinten      pin;    "HPI host interrupt enable
      tbcinten    pin;    "TBC host interrupt enable
      dspnmi      pin;    "DSP NMI interrupt from host
   "-------------------------------------------------------------------------------
   " From DSP-mapped Registers
      nmisel      pin;    "NMI interrupt source selection (host/codec)
      nmien       pin;    "NMI interrupt enable
      pcimren     pin;    "PCI master read interrupt (EXT_INT5) enable
      pcimwen     pin;    "PCI master write interrupt (EXT_INT6) enable
   "-------------------------------------------------------------------------------
   " From Memory Decode
      RDFIFO_     pin;    "Read PCI controller FIFO strobe
      WRFIFO_     pin;    "Write PCI controller FIFO strobe
"OUTPUTS
   "-------------------------------------------------------------------------------
   " To PCI Controller
      PC_INT      pin istype 'reg_d, dc, buffer';   "Add-on to PCI interrupt
   "-------------------------------------------------------------------------------
   " To DSP
      PCI_INT     pin istype 'com';                 "PCI-to-DSP interrupt (EXT_INT4)
      NMI         pin istype 'com';                 "NMI interrupt to DSP
      PCIMRD_INT  pin istype 'reg_d, dc, buffer';   "PCI master read int to DSP
      PCIMWR_INT  pin istype 'reg_d, dc, buffer';   "PCI master write int to DSP
"Internal Registers
      "DSP host interrupt falling edge detectors
      hint_db1, hint_db2        node istype 'reg_d, dc, buffer';
      "TBC interrupt falling edge detectors
      tbcint_db1, tbcint_db2    node istype 'reg_d, dc, buffer';
```

```
        "PCI master write state machine
        mwr_fifo1, mwr_fifo0      node istype 'reg_d, dc, buffer';
        mwr_fifo = [mwr_fifo1..mwr_fifo0];

        "PCI master read state machine
        mrd_fifo1, mrd_fifo0      node istype 'reg_d, dc, buffer';
        mrd_fifo = [mrd_fifo1..mrd_fifo0];

"Constants

    h,l,x = 1, 0, .X. ;

    "PCI master read state machine states
        MRD_IDLE        = [ 0, 0 ];          "wait for read FIFO not empty
        MRD_INT         = [ 0, 1 ];          "interrupt DSP (EXT_INT5)
        MRD_WAIT        = [ 1, 0 ];          "wait for EMIF read FIFO access
        MRD_DONE        = [ 1, 1 ];          "read FIFO transfer complete

    "PCI master write state machine states
        MWR_IDLE        = [ 0, 0 ];          "wait for write FIFO not full
        MWR_INT         = [ 0, 1 ];          "interrupt DSP (EXT_INT6)
        MWR_WAIT        = [ 1, 0 ];          "wait for EMIF write FIFO access
        MWR_DONE        = [ 1, 1 ];          "write FIFO transfer complete

Equations

    "-------------------------------------------------------------------------------
    "Interrupt to Host (INTA#)
    "-------------------------------------------------------------------------------
        "Falling edge detection of DSP host interrupt
        hint_db1    := DSP_HINT_;
        hint_db2    := hint_db1;
        hint_db1.ar  = RESET;
        hint_db2.ar  = RESET;
        hint_db1.clk = PCICLK;
        hint_db2.clk = PCICLK;

        "Falling edge detection of TBC interrupt
        tbcint_db1    := TBC_INT_;
        tbcint_db2    := tbcint_db1;
        tbcint_db1.ar  = RESET;
        tbcint_db2.ar  = RESET;
        tbcint_db1.clk = PCICLK;
        tbcint_db2.clk = PCICLK;

        "PC interrupt rising edge whenever DSP host interrupt
        "or TBC interrupt have a falling edge
        PC_INT        :=  !((tbcinten & !tbcint_db1 & tbcint_db2) #
                            (hinten   & !hint_db1   & hint_db2));
```

```
   PC_INT.ar      = RESET;
   PC_INT.clk     = PCICLK;

"-------------------------------------------------------------------------------
"NMI interrupt (from host or CODEC)
"-------------------------------------------------------------------------------
   "DSP selects NMI source of host (0) or (1) with nmisel register bit.
   "NMI is only generated when the DSP enables it with the nmien register bit.
   NMI = nmien & ((!nmisel & dspnmi) # (nmisel & CODEC_IRQ));

"-------------------------------------------------------------------------------
"PCI interrupt to DSP
"-------------------------------------------------------------------------------
   "Invert polarity of PCI controller's IRQ# output.
   "Only active when in PCI slot and not in reset
   PCI_INT = !PCI_DET_ & !PCI_IRQ_ & !RESET;

"-------------------------------------------------------------------------------
" PCI master read control
"-------------------------------------------------------------------------------
   "Master read interrupt register definition (controls DSP's EXT_INT5)
   PCIMRD_INT.ar  = RESET;         "Default to EXT_INT5 low
   PCIMRD_INT.clk = PCICLK;
   PCIMRD_INT.ce  = !PCI_DET_;     "Only activate when in PCI slot

   "Master read control state machine definition
   mrd_fifo.ar    = RESET;         "Default to MRD_IDLE state
   mrd_fifo.clk   = PCICLK;
   mrd_fifo.ce    = !PCI_DET_;     "Only activate when in PCI slot

   "Master read control state machine (mrd_fifo)
   state_diagram mrd_fifo

      "Wait for PCI master read state machine to be enabled (pcimren=1)
      "and PCI controller's read FIFO to not be empty.
      state MRD_IDLE:
         PCIMRD_INT := 0;
         if (pcimren & !RDEMPTY) then MRD_INT;
         else MRD_IDLE;

      "PCI master read state machine is enabled and PCI controller's
      "read FIFO is not empty.  Assert DSP's EXT_INT5 interrupt high.
      state MRD_INT:
         PCIMRD_INT := 1;     "set EXT_INT5 high
         goto MRD_WAIT;
```

```
            ”Wait for DSP to read FIFO at 0x1310000 or 0x1710000 or
            ”DSP to disable PCI master read state machine (pcimren=0).
            ”EXT_INT5 is held high during this state.
            state MRD_WAIT:
               PCIMRD_INT := 1;    ”wait for master read disable
                                   ”or start of FIFO read
               if (!pcimren # !RDFIFO_) then MRD_DONE;
               else MRD_WAIT;

            ”DSP has read PCI FIFO or disabled the PCI master read state machine.
            ”Return to idle state.
            state MRD_DONE:
               PCIMRD_INT := 1;    ”wait for master read disable
                                   ”or end of FIFO read
               if (!pcimren # RDFIFO_) then MRD_IDLE;
               else MRD_DONE;

Equations

   ”-------------------------------------------------------------------------------
   ” PCI master write control
   ”-------------------------------------------------------------------------------
      ”Master write interrupt register definition (controls DSP’s EXT_INT6)
      PCIMWR_INT.ar  = RESET;         ”Default to EXT_INT6 low
      PCIMWR_INT.clk = PCICLK;
      PCIMWR_INT.ce  = !PCI_DET_;     ”Only activate when in PCI slot

      ”Master write control state machine definition
      mwr_fifo.ar    = RESET;         ”Default to MWR_IDLE state
      mwr_fifo.clk   = PCICLK;
      mwr_fifo.ce    = !PCI_DET_;     ”Only activate when in PCI slot

      state_diagram mwr_fifo

         ”Wait for PCI master write state machine to be enabled (pcimwen=1)
         ”and PCI controller’s write FIFO to not full.
         state MWR_IDLE:
            PCIMWR_INT := 0;
            if (pcimwen & !WRFULL) then MWR_INT;
            else MWR_IDLE;

         ”PCI master write state machine is enabled and PCI controller’s
         ”write FIFO is not full.  Assert DSP’s EXT_INT6 interrupt high.
         state MWR_INT:
            PCIMWR_INT := 1;     ”set EXT_INT6 high
            goto MWR_WAIT;
```

```
            ″Wait for DSP to write FIFO at 0x1310000 or 0x1710000 or
            ″DSP to disable PCI master write state machine (pcimwen=0).
            ″EXT_INT6 is held high during this state.
            state MWR_WAIT:
               PCIMWR_INT := 1;    ″wait for master write disable
                                   ″or start of FIFO write

               if (!pcimwen # !WRFIFO_) then MWR_DONE;
               else MWR_WAIT;

            ″DSP has written PCI FIFO or disabled the PCI master write state machine.
            ″Return to idle state.
            state MWR_DONE:
               PCIMWR_INT := 1;    ″wait for master write disable
                                   ″or end of FIFO write
               if (!pcimwen # WRFIFO_) then MWR_IDLE;
               else MWR_DONE;

END
```

```
MODULE EVMPCI

interface (RESET,ARE_,AWE_,BE3_,BE2_,BE1_,BE0_,EA16,EA6..EA2,
          PCI_DET_,PCICLK,PTATN_,PTBURST_,PTNUM1,PTNUM0,PTWR,
          pt_be3_,pt_be2_,pt_be1_,pt_be0_,
          pt_addr4,pt_addr3,pt_addr2,pt_addr1, pt_addr0,
          TBC_RDY_,DSP_HRDY_,emif_req ->
          PCI_FLT_,PT_ADR_,PT_RDY_,AO_SEL_,AO_WR_,AO_RD_,
          AO_ADR6..AO_ADR2,AO_BE3_,AO_BE2_,AO_BE1_,AO_BE0_,RDFIFO_,WRFIFO_,
          pcireg_oe,pcireg_ce,TBC_WR_,TBC_RD_,
          HCS_,HDS1_,HRW,TBCA4_HCNTL1,TBCA3_HCNTL0,TBCA2_HHWIL,
          TBCA1_HBE1_,TBCA0_HBE0_,ao_bsy,emif_ack);

TITLE 'TMS320C6x EVM PCI Controller Logic'

"DWG NAME  TMS320C6x Evaluation Module (EVM)
"ASSY#     D600830-0001
"PAL #     U12, 830*
"PAL TYPE  Altera EPM7256SQC208-10 (256-Macrocells, 10-ns, 208-PQFP)
"COMPANY   Texas Instruments Incorporated / DNA Enterprises, Inc.
"ENGR      Brian G. Carlson
"DATE      02/21/98
"TOOLS     Synario ABEL 6.5 / Altera Max Plus II 8.2

"DESCRIPTION
   "This module controls the slave (target) interface between the
   "PCI controller and the JTAG TBC, DSP HPI and CPLD PCI registers,
   "as well as DSP EMIF accesses to the PCI controller which include
   "PCI bus master read/write tranfers.  This module consists of a
   "large state machine that monitors and controls all the signals
   "that are required to transfer data between the PCI controller and
   "the devices on the 'C6x EVM board.  The PCI controller handles
   "all the PCI bus transactions, so the CPLD is not involved at all
   "on the actual PCI bus.  The CPLD only manages the transfers with
   "the PCI controller's add-on bus.  The state machine in this module
   "runs from the buffered PCI clock which has a maximum frequency of
   "33 MHz.

"REVISIONS
"   04/15/98 - Modified HPI write timing to sample HRDY- before raising
"              HDS1- instead of at beginning of cycle.  The previous
"              TMS320C62x Peripherals Reference Guide dated Oct 1997
"              stated on page 5-8 that HRDY- was inactive (high) when
"              HCS- was inactive.  However, HRDY- is actually active (low)
"              when HCS- is inactive, so HPI writes previously got an
"              incorrect ready indication and never waited for a valid
"              ready indication with a potential for missed HPI writes.
```

```
"
"   03/11/98 – Modified HPI write timing to work with new 'C6201 data
"              sheet timing which samples HBE[1:0] on the rising edge of
"              HSTROBE–, instead of falling edge.  This was not a problem
"              with Rev. 2.0 silicon.  However, the BE2– and BE3– signals
"              do not get generated with Rev. 2.1 silicon without this
"              change.
"
"   03/05/98 – Modified DSP EMIF ready logic for PCI controller registers
"              and FIFO accesses to hold ARDY active until ARE– or AWE–
"              is deasserted.  This allows the PCI controller to be
"              accessed with any strobe period >= 3 CLKOUT1 clocks.

"INPUTS
   "-------------------------------------------------------------------------------
   " From Voltage Supervisor
      RESET       pin;    "Active–high board reset
   "-------------------------------------------------------------------------------
   " From TMS320C6201 DSP
      ARE_        pin;    "EMIF asynchronous memory read strobe
      AWE_        pin;    "EMIF asynchronous memory write strobe
      BE3_        pin;    "EMIF byte 3 enable control
      BE2_        pin;    "EMIF byte 2 enable control
      BE1_        pin;    "EMIF byte 1 enable control
      BE0_        pin;    "EMIF byte 0 enable control
      BE = [BE3_, BE2_, BE1_, BE0_];
      "EMIF word address bits [16], [6..2]
      EA16        pin;
      EA6         pin;
      EA5         pin;
      EA4         pin;
      EA3         pin;
      EA2         pin;
   "-------------------------------------------------------------------------------
   " From S5933 PCI Controller
      PCI_DET_    pin;    "PCI bus detection (0=PCI, 1=Standalone)
      PCICLK      pin;    "Buffered PCI clock (33 MHz max.)
      PTATN_      pin;    "Pass–thru attention indication
      PTBURST_    pin;    "Pass–thru burst indication
      PTNUM1      pin;    "Pass–thru region number (bit 1)
      PTNUM0      pin;    "Pass–thru region number (bit 0)
      PTWR        pin;    "Pass–thru access type (R=0, W=1)
```

```
    pt_be3_     pin;    "Latched pass-thru byte enables [3..0]
    pt_be2_     pin;
    pt_be1_     pin;
    pt_be0_     pin;
    pt_addr4    pin;    "Latched pass-thru address[4..0]
    pt_addr3    pin;
    pt_addr2    pin;
    pt_addr1    pin;
    pt_addr0    pin;

  "-------------------------------------------------------------------------------
  " From JTAG TBC
    TBC_RDY_    pin;    "JTAG TBC ready
  "-------------------------------------------------------------------------------
  " From DSP HPI
    DSP_HRDY_   pin;    "DSP HPI ready
  "-------------------------------------------------------------------------------
  " From Memory Decode
    emif_req    pin;    "Synchronized EMIF access request
  "-------------------------------------------------------------------------------

"OUTPUTS
  "-------------------------------------------------------------------------------
  " To S5933 PCI Controller
    PCI_FLT_    pin istype 'com';               "Float PCI controller outputs
    PT_ADR_     pin istype 'reg_d, dc, buffer'; "Pass-thru address request
    PT_RDY_     pin istype 'reg_d, dc, buffer'; "Pass-thru ready indication
    AO_SEL_     pin istype 'reg_d, dc, buffer'; "Add-on select for register access
    AO_WR_      pin istype 'reg_d, dc, buffer'; "Add-on write strobe
    AO_RD_      pin istype 'reg_d, dc, buffer'; "Add-on read strobe

    "Add-on address bits [6..2]
    AO_ADR6     pin istype 'reg_d, dc, buffer';
    AO_ADR5     pin istype 'reg_d, dc, buffer';
    AO_ADR4     pin istype 'reg_d, dc, buffer';
    AO_ADR3     pin istype 'reg_d, dc, buffer';
    AO_ADR2     pin istype 'reg_d, dc, buffer';
    AO_ADR = [AO_ADR6..AO_ADR2];

    "Add-on byte enables [3..0]
    AO_BE3_     pin istype 'reg_d, dc, buffer';
    AO_BE2_     pin istype 'reg_d, dc, buffer';
    AO_BE1_     pin istype 'reg_d, dc, buffer';
    AO_BE0_     pin istype 'reg_d, dc, buffer';
```

```
    RDFIFO_     pin istype 'reg_d, dc, buffer';  "Read FIFO strobe
    WRFIFO_     pin istype 'reg_d, dc, buffer';  "Write FIFO strobe

  "-------------------------------------------------------------------------------
  " To PCI Memory-mapped CPLD Registers
    pcireg_oe   pin istype 'reg_d, dc, buffer';  "PCI register output enable
    pcireg_ce   pin istype 'reg_d, dc, buffer';  "PCI register clock enable

  "-------------------------------------------------------------------------------
  " To JTAG TBC
    TBC_WR_     pin istype 'reg_d, dc, buffer';  "TBC write strobe
    TBC_RD_     pin istype 'reg_d, dc, buffer';  "TBC read strobe

  "-------------------------------------------------------------------------------
  " To DSP HPI
    HCS_        pin  istype 'reg_d, dc, buffer'; "HPI chip select
    HDS1_       pin  istype 'reg_d, dc, buffer'; "HPI data strobe
    HRW         pin  istype 'reg_d, dc, buffer'; "HPI read/write control

  "-------------------------------------------------------------------------------
  " To TBC and HPI
    TBCA4_HCNTL1    pin istype 'reg_d, dc,buffer';  "TBC addr / HPI byte enable 0
    TBCA3_HCNTL0    pin istype 'reg_d, dc,buffer';  "TBC addr 1 / HPI byte enable 1
    TBCA2_HHWIL     pin istype 'reg_d, dc,buffer';  "TBC addr 2 / HPI half-word sel
    TBCA1_HBE1_     pin istype 'reg_d, dc,buffer';  "TBC addr 3 / HPI control 0
    TBCA0_HBE0_     pin istype 'reg_d, dc,buffer';  "TBC addr 4 / HPI control 1

  "Add-on busy flag
    ao_bsy      pin istype 'reg_d, dc, buffer';  "Add-on state machine busy flag
    emif_ack    pin istype 'reg_d, dc, buffer';  "EMIF access acknowledge

"Internal Registers

  "Add-on state machine
  "Implemented with 15 bits to limit product terms and ease routing.
    aos14, aos13, aos12, aos11, aos10,
    aos9,  aos8,  aos7,  aos6,  aos5,
    aos4,  aos3,  aos2,  aos1,  aos0   node istype 'reg_d, dc, buffer';
    aos = [aos14..aos0];

  "TBC ready flag
    tbcrdy      node istype 'reg_d, dc, buffer';

  "HPI ready flag (synchronized)
    hrdy        node istype 'reg_d, dc, buffer';

  "HPI control [1..0]
    hcntl1      node istype 'com';
    hcntl0      node istype 'com';
```

```
"Constants

   h,l,x = 1, 0, .X. ;

   "Latched pass-thru address bits
   pt_addr = [pt_addr4..pt_addr0] ;

   "PCI Controller BAR decode.  The S5933 controls the PTNUM[1..0] signals
   "to indicate which base region is being accessed.
      BAR1          = !PTNUM1 & !PTNUM0;       "JTAG TBC register access
      BAR2          = !PTNUM1 &  PTNUM0;       "CPLD register access
      BAR3          =  PTNUM1 & !PTNUM0;       "DSP HPI register access
      BAR4          =  PTNUM1 &  PTNUM0;       "DSP HPI auto-inc data access

   "PCI Pass-thru Access Indications
      TBC_WR        = !PTATN_ &  PTWR & BAR1;  "JTAG TBC register write
      TBC_RD        = !PTATN_ & !PTWR & BAR1;  "JTAG TBC register read
      REG_WR        = !PTATN_ &  PTWR & BAR2;  "CPLD register write
      REG_RD        = !PTATN_ & !PTWR & BAR2;  "CPLD register read
      HPI_WR        = !PTATN_ &  PTWR & BAR3;  "DSP HPI register write
      HPI_RD        = !PTATN_ & !PTWR & BAR3;  "DSP HPI register read
      HPI_BURST_WR  = !PTATN_ &  PTWR & BAR4;  "DSP HPI auto-inc data write
      HPI_BURST_RD  = !PTATN_ & !PTWR & BAR4;  "DSP HPI auto-inc data read

   "PCI add-on control register group
      ao_cntl     =[PT_ADR_, AO_SEL_, AO_WR_, AO_RD_,
                   AO_ADR6, AO_ADR5, AO_ADR4, AO_ADR3, AO_ADR2,
                   AO_BE3_, AO_BE2_, AO_BE1_, AO_BE0_,
                   RDFIFO_, WRFIFO_, PT_RDY_,
                   pcireg_ce, pcireg_oe];

   "=============================================================================
   "PCI add-on control register group vectors
   "-----------------------------------------------------------------------------
   "A add-on control vector is output to the S5933 PCI controller at each
   "state of the defined accesses.  Since the state machine executes at
   "the PCI clock rate of 33 MHz, each state lasts a minimum of 30 ns.
   "-----------------------------------------------------------------------------
      "              P A A A A A A A A A A A A R W P p p
      "              T 0 0 0 0 0 0 0 0 0 0 0 0 F F T c c
      "              _ _ _ _ _ _ _ _ _ _ _ _ I I _ i i
      "              A S W R A A A A A B B B B F F R r r
      "              D E R D D D D D D E E E E 0 0 D e e
      "              R L _ _ R R R R R 3 2 1 0 _ _ Y g g
      "              _ _     6 5 4 3 2 _ _ _ _     _ _ _
      "                                            c o
      "                                            e e
```

```
"-------------------------------------------------------------------------------

    "Idle
    AOC_IDLE1   = [1,1,1,1,0,0,0,0,0,1,1,1,1,1,1,1,0,0]; "Wait for PCI/EMIF access
    AOC_IDLE2   = [1,1,1,1,0,0,0,0,0,1,1,1,1,1,1,1,0,0]; "EMIF access
    AOC_IDLE3   = [0,0,1,1,0,1,0,1,1,1,1,1,1,1,1,1,0,0]; "PCI access
    AOC_IDLE4   = [1,0,1,0,0,1,0,1,1,1,1,1,1,1,1,1,0,0]; "BAR4 write
    AOC_IDLE5   = [1,0,0,1,0,1,0,1,1,1,1,1,1,1,1,1,0,0]; "BAR4 read

    "TBC Write
    AOC_TBC_WR1 = [1,0,1,0,0,1,0,1,1,0,0,0,0,1,1,1,0,0];
    AOC_TBC_WR2 = [1,0,1,0,0,1,0,1,1,0,0,0,0,1,1,1,0,0];
    AOC_TBC_WR3 = [1,0,1,0,0,1,0,1,1,0,0,0,0,1,1,1,0,0];
    AOC_TBC_WR4 = [1,0,1,1,0,1,0,1,1,1,1,1,1,1,1,0,0,0];

    "TBC Read
    AOC_TBC_RD1 = [1,0,0,1,0,1,0,1,1,0,0,0,0,1,1,1,0,0];
    AOC_TBC_RD2 = [1,0,0,1,0,1,0,1,1,0,0,0,0,1,1,1,0,0];
    AOC_TBC_RD3 = [1,0,0,1,0,1,0,1,1,0,0,0,0,1,1,1,0,0]
    AOC_TBC_RD4 = [1,0,1,1,0,1,0,1,1,1,1,1,1,1,1,0,0,0];

    "PCI-mapped Register Write
    AOC_REG_WR1 = [1,0,1,0,0,1,0,1,1,0,0,0,0,1,1,1,0,0];
    AOC_REG_WR2 = [1,0,1,0,0,1,0,1,1,0,0,0,0,1,1,1,1,0];
    AOC_REG_WR3 = [1,0,1,1,0,1,0,1,1,1,1,1,1,1,1,0,0,0];

    "PCI-mapped Register Read
    AOC_REG_RD1 = [1,0,1,1,0,1,0,1,1,0,0,0,0,1,1,1,0,0];
    AOC_REG_RD2 = [1,0,0,1,0,1,0,1,1,0,0,0,0,1,1,1,0,1];
    AOC_REG_RD3 = [1,0,1,1,0,1,0,1,1,1,1,1,1,1,1,0,0,1];

    "HPI Write (BAR3)
    AOC_HPI_WR1 = [1,0,1,0,0,1,0,1,1,1,1,1,0,0,1,1,1,0,0];
    AOC_HPI_WR2 = [1,0,1,0,0,1,0,1,1,1,1,1,0,0,1,1,1,0,0];
    AOC_HPI_WR3 = [1,0,1,0,0,1,0,1,1,1,1,1,0,0,1,1,1,0,0];
    AOC_HPI_WR4 = [1,0,1,0,0,1,0,1,1,1,1,1,0,0,1,1,1,0,0];
    AOC_HPI_WR5 = [1,0,1,0,0,1,0,1,1,0,0,1,1,1,1,1,1,0,0];
    AOC_HPI_WR6 = [1,0,1,0,0,1,0,1,1,0,0,1,1,1,1,1,1,0,0];
    AOC_HPI_WR7 = [1,0,1,0,0,1,0,1,1,0,0,1,1,1,1,1,1,0,0];
    AOC_HPI_WR8 = [1,0,1,1,0,1,0,1,1,1,1,1,1,1,1,0,0,0];
```

```
"HPI Read (BAR3)
AOC_HPI_RD1 = [1,0,0,1,0,1,0,1,1,1,1,1,1,1,1,1,0,0];
AOC_HPI_RD2 = [1,0,0,1,0,1,0,1,1,1,1,1,1,1,1,1,0,0];
AOC_HPI_RD3 = [1,0,0,1,0,1,0,1,1,1,1,0,0,1,1,1,0,0];
AOC_HPI_RD4 = [1,0,0,1,0,1,0,1,1,1,1,0,0,1,1,1,0,0];
AOC_HPI_RD5 = [1,0,0,1,0,1,0,1,1,1,1,0,0,1,1,1,0,0];
AOC_HPI_RD6 = [1,0,0,1,0,1,0,1,1,0,0,1,1,1,1,1,0,0];
AOC_HPI_RD7 = [1,0,0,1,0,1,0,1,1,0,0,1,1,1,1,1,0,0];
AOC_HPI_RD8 = [1,0,1,1,0,1,0,1,1,1,1,1,1,1,1,0,0,0];

"HPI Burst Write (BAR4)
AOC_HPI_BWR1= [1,0,1,0,0,1,0,1,1,1,1,1,1,1,1,1,0,0];
AOC_HPI_BWR2= [1,0,1,0,0,1,0,1,1,1,1,0,0,1,1,1,0,0];
AOC_HPI_BWR3= [1,0,1,0,0,1,0,1,1,1,1,0,0,1,1,1,0,0];
AOC_HPI_BWR4= [1,0,1,0,0,1,0,1,1,1,1,0,0,1,1,1,0,0];
AOC_HPI_BWR5= [1,0,1,0,0,1,0,1,1,0,0,1,1,1,1,1,0,0];
AOC_HPI_BWR6= [1,0,1,0,0,1,0,1,1,0,0,1,1,1,1,0,0,0];
AOC_HPI_BWR7= [1,1,1,1,0,1,0,1,1,1,1,1,1,1,1,1,0,0];
AOC_HPI_BWR8= [1,1,1,1,0,1,0,1,1,1,1,1,1,1,1,1,0,0];

"HPI Burst Read (BAR4)
AOC_HPI_BRD1= [1,0,0,1,0,1,0,1,1,1,1,1,1,1,1,1,0,0];
AOC_HPI_BRD2= [1,0,0,1,0,1,0,1,1,1,1,0,0,1,1,1,0,0];
AOC_HPI_BRD3= [1,0,0,1,0,1,0,1,1,1,1,0,0,1,1,1,0,0];
AOC_HPI_BRD4= [1,0,0,1,0,1,0,1,1,1,1,0,0,1,1,1,0,0];
AOC_HPI_BRD5= [1,0,0,1,0,1,0,1,1,0,0,1,1,1,1,1,0,0];
AOC_HPI_BRD6= [1,0,0,1,0,1,0,1,1,0,0,1,1,1,1,1,0,0];
AOC_HPI_BRD7= [1,0,1,1,0,1,0,1,1,1,1,1,1,1,1,0,0,0];
AOC_HPI_BRD8= [1,1,1,1,0,1,0,1,1,1,1,1,1,1,1,1,0,0];
AOC_HPI_BRD9= [1,1,1,1,0,1,0,1,1,1,1,1,1,1,1,1,0,0];

"EMIF Write to PCI Controller Register
AOC_EMIF_WR1= [1,0,0,1,EA6,EA5,EA4,EA3,EA2,BE3_,BE2_,BE1_,BE0_,1,1,1,0,0];
AOC_EMIF_WR2= [1,0,1,1,EA6,EA5,EA4,EA3,EA2,BE3_,BE2_,BE1_,BE0_,1,1,1,0,0];
AOC_EMIF_WR3= [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0];

"EMIF Read from PCI Controller Register
AOC_EMIF_RD1= [1,0,1,0,EA6,EA5,EA4,EA3,EA2,BE3_,BE2_,BE1_,BE0_,1,1,1,0,0];
AOC_EMIF_RD2= [1,0,1,0,EA6,EA5,EA4,EA3,EA2,BE3_,BE2_,BE1_,BE0_,1,1,1,0,0];

"EMIF Write to FIFO
AOC_FIFO_WR1= [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,0,0];
AOC_FIFO_WR2= [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0];

"EMIF Read from FIFO
AOC_FIFO_RD1= [1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,0,0];
AOC_FIFO_RD2= [1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,0,0];
AOC_FIFO_RD3= [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0];
```

```
   ″Done
   AOC_DONE    = [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0];

″TBC/HPI control register group
   th_cntl= [TBC_WR_,TBC_RD_,HCS_,HDS1_,
             TBCA4_HCNTL1,TBCA3_HCNTL0,TBCA2_HHWIL,
             TBCA1_HBE1_,TBCA0_HBE0_];

″=============================================================================
″TBC/HPI control register group vectors
″-----------------------------------------------------------------------------
″A TBC/HPI control vector is output to the JTAG TBC and DSP HPI at each
″state of the defined accesses.  Since the state machine executes at
″the PCI clock rate of 33 MHz, each state lasts a minimum of 30 ns.
″-----------------------------------------------------------------------------
   ″              T T H H T T T T T
   ″              B B C D B B B B B
   ″              C C S S C C C C C
   ″              _ _ _ 1 A A A A A
   ″              W R   _ 4 3 2 1 0
   ″              R D     _ _ _ _ _
   ″              _ _       H H H H H
   ″                        C C H B B
   ″                        N N W E E
   ″                        T T I 1 0
   ″                        L L L _ _
   ″                        1 0
″-----------------------------------------------------------------------------
″Idle
   THC_IDLE1   = [1,1,1,1,1,1,1,1,1];
   THC_IDLE2   = [1,1,1,1,1,1,1,1,1];
   THC_IDLE3   = [1,1,1,1,1,1,1,1,1];

   ″TBC Write
   THC_TBC_WR1 = [1,1,1,1,pt_addr4..pt_addr0];
   THC_TBC_WR2 = [0,1,1,1,pt_addr4..pt_addr0];
   THC_TBC_WR3 = [1,1,1,1,pt_addr4..pt_addr0];
   THC_TBC_WR4 = [1,1,1,1,1,1,1,1,1];

   ″TBC Read
   THC_TBC_RD1 = [1,1,1,1,pt_addr4..pt_addr0];
   THC_TBC_RD2 = [1,0,1,1,pt_addr4..pt_addr0];
   THC_TBC_RD3 = [1,0,1,1,pt_addr4..pt_addr0];
   THC_TBC_RD4 = [1,0,1,1,pt_addr4..pt_addr0];
```

```
"PCI-mapped Register Write
THC_REG_WR1 = [1,1,1,1,1,1,1,1,1];
THC_REG_WR2 = [1,1,1,1,1,1,1,1,1];
THC_REG_WR3 = [1,1,1,1,1,1,1,1,1];

"PCI-mapped Register Read
THC_REG_RD1 = [1,1,1,1,1,1,1,1,1];
THC_REG_RD2 = [1,1,1,1,1,1,1,1,1];
THC_REG_RD3 = [1,1,1,1,1,1,1,1,1];

"HPI Write (BAR3)
THC_HPI_WR1 = [1,1,0,1,1,1,1,1,1];
THC_HPI_WR2 = [1,1,0,1,hcntl1,hcntl0,0,pt_be1_,pt_be0_];
THC_HPI_WR3 = [1,1,0,0,hcntl1,hcntl0,0,pt_be1_,pt_be0_];
THC_HPI_WR4 = [1,1,0,0,hcntl1,hcntl0,0,pt_be1_,pt_be0_];
THC_HPI_WR5 = [1,1,0,1,hcntl1,hcntl0,1,pt_be1_,pt_be0_];
THC_HPI_WR6 = [1,1,0,0,hcntl1,hcntl0,1,pt_be3_,pt_be2_];
THC_HPI_WR7 = [1,1,0,1,1,1,1,pt_be3_,pt_be2_];
THC_HPI_WR8 = [1,1,1,1,1,1,1,1,1];


"HPI Read (BAR3)
THC_HPI_RD1 = [1,1,0,1,1,1,1,1,1];
THC_HPI_RD2 = [1,1,0,1,hcntl1,hcntl0,0,1,1];
THC_HPI_RD3 = [1,1,0,0,hcntl1,hcntl0,0,1,1];
THC_HPI_RD4 = [1,1,0,0,hcntl1,hcntl0,0,1,1];
THC_HPI_RD5 = [1,1,0,0,hcntl1,hcntl0,0,1,1];
THC_HPI_RD6 = [1,1,0,1,hcntl1,hcntl0,1,1,1];
THC_HPI_RD7 = [1,1,0,0,hcntl1,hcntl0,1,1,1];
THC_HPI_RD8 = [1,1,0,1,1,1,1,1,1];

"HPI Burst Write (BAR4)
THC_HPI_BWR1= [1,1,0,1,1,0,0,0,0];
THC_HPI_BWR2= [1,1,0,0,1,0,0,0,0];
THC_HPI_BWR3= [1,1,0,0,1,0,0,0,0];
THC_HPI_BWR4= [1,1,0,1,1,0,1,0,0];
THC_HPI_BWR5= [1,1,0,0,1,0,1,0,0];
THC_HPI_BWR6= [1,1,0,1,1,1,1,0,0];
THC_HPI_BWR7= [1,1,1,1,1,1,1,1,1];
THC_HPI_BWR8= [1,1,1,1,1,1,1,1,1];

"HPI Burst Read (BAR4)
THC_HPI_BRD1= [1,1,0,1,1,0,0,1,1];
THC_HPI_BRD2= [1,1,0,0,1,0,0,1,1];
THC_HPI_BRD3= [1,1,0,0,1,0,0,1,1];
THC_HPI_BRD4= [1,1,0,0,1,0,0,1,1];
THC_HPI_BRD5= [1,1,0,1,1,0,1,1,1];
```

```
    THC_HPI_BRD6= [1,1,0,0,1,0,1,1,1];
    THC_HPI_BRD7= [1,1,0,1,1,1,1,1,1];
    THC_HPI_BRD8= [1,1,1,1,1,1,1,1,1];
    THC_HPI_BRD9= [1,1,1,1,1,1,1,1,1];

    "EMIF Write to Register
    THC_EMIF_WR1= [1,1,1,1,1,1,1,1,1];
    THC_EMIF_WR2= [1,1,1,1,1,1,1,1,1];
    THC_EMIF_WR3= [1,1,1,1,1,1,1,1,1];

    "EMIF Read from Register
    THC_EMIF_RD1= [1,1,1,1,1,1,1,1,1];
    THC_EMIF_RD2= [1,1,1,1,1,1,1,1,1];

    "EMIF Write to FIFO
    THC_FIFO_WR1= [1,1,1,1,1,1,1,1,1];
    THC_FIFO_WR2= [1,1,1,1,1,1,1,1,1];

    "EMIF Read from FIFO
    THC_FIFO_RD1= [1,1,1,1,1,1,1,1,1];
    THC_FIFO_RD2= [1,1,1,1,1,1,1,1,1];
    THC_FIFO_RD3= [1,1,1,1,1,1,1,1,1];

    "Done
    THC_DONE    = [1,1,1,1,1,1,1,1,1];

"Add-on state machine states
    "                   T  H  HB  R  E  F  RW   SUB-STATE              STATE
"-------------------------------------------------------------------------------
    AO_IDLE1        = [ 0, 0, 0,  0, 0, 0, 0,   0,0,0,0,0,0,0,0 ];   "0x0000
    AO_IDLE2        = [ 0, 0, 0,  0, 0, 0, 0,   0,0,0,0,0,0,0,1 ];   "0x0001
    AO_IDLE3        = [ 0, 0, 0,  0, 0, 0, 0,   0,0,0,0,0,0,1,0 ];   "0x0002
    AO_DONE         = [ 0, 0, 0,  0, 0, 0, 0,   1,0,0,0,0,0,0,0 ];   "0x0080

    TBC_WR1         = [ 1, 0, 0,  0, 0, 0, 1,   0,0,0,0,0,0,0,0 ];   "0x4100
    TBC_WR2         = [ 1, 0, 0,  0, 0, 0, 1,   0,0,0,0,0,0,0,1 ];   "0x4101
    TBC_WR3         = [ 1, 0, 0,  0, 0, 0, 1,   0,0,0,0,0,0,1,0 ];   "0x4102
    TBC_WR4         = [ 1, 0, 0,  0, 0, 0, 1,   0,0,0,0,0,1,0,0 ];   "0x4104

    TBC_RD1         = [ 1, 0, 0,  0, 0, 0, 0,   0,0,0,0,0,0,0,0 ];   "0x4000
    TBC_RD2         = [ 1, 0, 0,  0, 0, 0, 0,   0,0,0,0,0,0,0,1 ];   "0x4001
    TBC_RD3         = [ 1, 0, 0,  0, 0, 0, 0,   0,0,0,0,0,0,1,0 ];   "0x4002
    TBC_RD4         = [ 1, 0, 0,  0, 0, 0, 0,   0,0,0,0,0,1,0,0 ];   "0x4004
```

```
HPI_WR1         = [ 0, 1, 0,  0, 0, 0, 1,   0,0,0,0,0,0,0,0 ];   "0x2100
HPI_WR2         = [ 0, 1, 0,  0, 0, 0, 1,   0,0,0,0,0,0,0,1 ];   "0x2101
HPI_WR3         = [ 0, 1, 0,  0, 0, 0, 1,   0,0,0,0,0,0,1,0 ];   "0x2102
HPI_WR4         = [ 0, 1, 0,  0, 0, 0, 1,   0,0,0,0,0,1,0,0 ];   "0x2104
HPI_WR5         = [ 0, 1, 0,  0, 0, 0, 1,   0,0,0,0,1,0,0,0 ];   "0x2108
HPI_WR6         = [ 0, 1, 0,  0, 0, 0, 1,   0,0,0,1,0,0,0,0 ];   "0x2110
HPI_WR7         = [ 0, 1, 0,  0, 0, 0, 1,   0,0,1,0,0,0,0,0 ];   "0x2120
HPI_WR8         = [ 0, 1, 0,  0, 0, 0, 1,   0,1,0,0,0,0,0,0 ];   "0x2140


HPI_RD1         = [ 0, 1, 0,  0, 0, 0, 0,   0,0,0,0,0,0,0,0 ];   "0x2000
HPI_RD2         = [ 0, 1, 0,  0, 0, 0, 0,   0,0,0,0,0,0,0,1 ];   "0x2001
HPI_RD3         = [ 0, 1, 0,  0, 0, 0, 0,   0,0,0,0,0,0,1,0 ];   "0x2002
HPI_RD4         = [ 0, 1, 0,  0, 0, 0, 0,   0,0,0,0,0,1,0,0 ];   "0x2004
HPI_RD5         = [ 0, 1, 0,  0, 0, 0, 0,   0,0,0,0,1,0,0,0 ];   "0x2008
HPI_RD6         = [ 0, 1, 0,  0, 0, 0, 0,   0,0,0,1,0,0,0,0 ];   "0x2010
HPI_RD7         = [ 0, 1, 0,  0, 0, 0, 0,   0,0,1,0,0,0,0,0 ];   "0x2020
HPI_RD8         = [ 0, 1, 0,  0, 0, 0, 0,   0,1,0,0,0,0,0,0 ];   "0x2040

HPI_BURST_WR1   = [ 0, 1, 1,  0, 0, 0, 1,   0,0,0,0,0,0,0,0 ];   "0x3100
HPI_BURST_WR2   = [ 0, 1, 1,  0, 0, 0, 1,   0,0,0,0,0,0,0,1 ];   "0x3101
HPI_BURST_WR3   = [ 0, 1, 1,  0, 0, 0, 1,   0,0,0,0,0,0,1,0 ];   "0x3102
HPI_BURST_WR4   = [ 0, 1, 1,  0, 0, 0, 1,   0,0,0,0,0,1,0,0 ];   "0x3104
HPI_BURST_WR5   = [ 0, 1, 1,  0, 0, 0, 1,   0,0,0,0,1,0,0,0 ];   "0x3108
HPI_BURST_WR6   = [ 0, 1, 1,  0, 0, 0, 1,   0,0,0,1,0,0,0,0 ];   "0x3110
HPI_BURST_WR7   = [ 0, 1, 1,  0, 0, 0, 1,   0,0,1,0,0,0,0,0 ];   "0x3120
HPI_BURST_WR8   = [ 0, 1, 1,  0, 0, 0, 1,   0,1,0,0,0,0,0,0 ];   "0x3140

HPI_BURST_RD1   = [ 0, 1, 1,  0, 0, 0, 0,   0,0,0,0,0,0,0,0 ];   "0x3000
HPI_BURST_RD2   = [ 0, 1, 1,  0, 0, 0, 0,   0,0,0,0,0,0,0,1 ];   "0x3001
HPI_BURST_RD3   = [ 0, 1, 1,  0, 0, 0, 0,   0,0,0,0,0,0,1,0 ];   "0x3002
HPI_BURST_RD4   = [ 0, 1, 1,  0, 0, 0, 0,   0,0,0,0,0,1,0,0 ];   "0x3004
HPI_BURST_RD5   = [ 0, 1, 1,  0, 0, 0, 0,   0,0,0,0,1,0,0,0 ];   "0x3008
HPI_BURST_RD6   = [ 0, 1, 1,  0, 0, 0, 0,   0,0,0,1,0,0,0,0 ];   "0x3010
HPI_BURST_RD7   = [ 0, 1, 1,  0, 0, 0, 0,   0,0,1,0,0,0,0,0 ];   "0x3020
HPI_BURST_RD8   = [ 0, 1, 1,  0, 0, 0, 0,   0,1,0,0,0,0,0,0 ];   "0x3040
HPI_BURST_RD9   = [ 0, 1, 1,  0, 0, 0, 0,   1,0,0,0,0,0,0,0 ];   "0x3080

REG_WR1         = [ 0, 0, 0,  1, 0, 0, 1,   0,0,0,0,0,0,0,0 ];   "0x0900
REG_WR2         = [ 0, 0, 0,  1, 0, 0, 1,   0,0,0,0,0,0,0,1 ];   "0x0901
REG_WR3         = [ 0, 0, 0,  1, 0, 0, 1,   0,0,0,0,0,0,1,0 ];   "0x0902

REG_RD1         = [ 0, 0, 0,  1, 0, 0, 0,   0,0,0,0,0,0,0,0 ];   "0x0800
REG_RD2         = [ 0, 0, 0,  1, 0, 0, 0,   0,0,0,0,0,0,0,1 ];   "0x0801
REG_RD3         = [ 0, 0, 0,  1, 0, 0, 0,   0,0,0,0,0,0,1,0 ];   "0x0802
```

```
        EMIF_WR1        = [ 0, 0, 0,  0, 1, 0, 1,   0,0,0,0,0,0,0,0 ];   "0x0500
        EMIF_WR2        = [ 0, 0, 0,  0, 1, 0, 1,   0,0,0,0,0,0,0,1 ];   "0x0501
        EMIF_WR3        = [ 0, 0, 0,  0, 1, 0, 1,   0,0,0,0,0,0,1,0 ];   "0x0502

        EMIF_RD1        = [ 0, 0, 0,  0, 1, 0, 0,   0,0,0,0,0,0,0,0 ];   "0x0400
        EMIF_RD2        = [ 0, 0, 0,  0, 1, 0, 0,   0,0,0,0,0,0,0,1 ];   "0x0401

        FIFO_WR1        = [ 0, 0, 0,  0, 0, 1, 1,   0,0,0,0,0,0,0,0 ];   "0x0300
        FIFO_WR2        = [ 0, 0, 0,  0, 0, 1, 1,   0,0,0,0,0,0,0,1 ];   "0x0301

        FIFO_RD1        = [ 0, 0, 0,  0, 0, 1, 0,   0,0,0,0,0,0,0,0 ];   "0x0200
        FIFO_RD2        = [ 0, 0, 0,  0, 0, 1, 0,   0,0,0,0,0,0,0,1 ];   "0x0201
        FIFO_RD3        = [ 0, 0, 0,  0, 0, 1, 0,   0,0,0,0,0,0,1,0 ];   "0x0202

Equations

    "Assert PCI float when not in PCI slot
    PCI_FLT_ = !PCI_DET_;

    "PCI add-on bus output enable when pcireg_oe set to 1
    pcireg_oe.ar  = RESET;
    pcireg_oe.clk = PCICLK;

    "PCI add-on bus clock enable when pcireg_ce set to 1
    pcireg_ce.ar  = RESET;
    pcireg_ce.clk = PCICLK;

    "-------------------------------------------------------------------------------
    " PCI Add-on Bus Control
    "-------------------------------------------------------------------------------
    "Add-on bus control signals

        "Default active-low control signals to high at reset
        PT_ADR_.ap = RESET;
        PT_RDY_.ap = RESET;
        AO_SEL_.ap = RESET;
        AO_WR_.ap  = RESET;
        AO_RD_.ap  = RESET;
        AO_ADR.ar  = RESET;
        AO_BE3_.ap = RESET;
        AO_BE2_.ap = RESET;
        AO_BE1_.ap = RESET;
        AO_BE0_.ap = RESET;
        RDFIFO_.ap = RESET;
        WRFIFO_.ap = RESET;
```

```
    "All registers are clocked by PCI clock
    AO_SEL_.clk= PCICLK;
    AO_WR_.clk = PCICLK;
    AO_RD_.clk = PCICLK;
    PT_ADR_.clk= PCICLK;
    PT_RDY_.clk= PCICLK;
    AO_ADR.clk = PCICLK;
    AO_BE3_.clk= PCICLK;
    AO_BE2_.clk= PCICLK;
    AO_BE1_.clk= PCICLK;
    AO_BE0_.clk= PCICLK;
    RDFIFO_.clk= PCICLK;
    WRFIFO_.clk= PCICLK;

"Add-on busy flag
    ao_bsy.ar  = RESET;
    ao_bsy.clk = PCICLK;

"EMIF acknowledgement flag
    emif_ack.ar  = RESET;
    emif_ack.clk = PCICLK;

"-------------------------------------------------------------------------------
" JTAG TBC / DSP HPI Control
"-------------------------------------------------------------------------------
"TBC control signals

    "Default active-low control signals to high at reset
    TBC_WR_.ap      = RESET;
    TBC_RD_.ap      = RESET;
    TBCA4_HCNTL1.ap = RESET;
    TBCA3_HCNTL0.ap = RESET;
    TBCA2_HHWIL.ap  = RESET;
    TBCA1_HBE1_.ap  = RESET;
    TBCA0_HBE0_.ap  = RESET;

    "All registers are clocked by PCI clock
    TBC_WR_.clk     = PCICLK;
    TBC_RD_.clk     = PCICLK;
    TBCA4_HCNTL1.clk= PCICLK;
    TBCA3_HCNTL0.clk= PCICLK;
    TBCA2_HHWIL.clk = PCICLK;
    TBCA1_HBE1_.clk = PCICLK;
    TBCA0_HBE0_.clk = PCICLK;
```

```
   "TBC ready synchronization
      tbcrdy     := !TBC_RDY_;
      tbcrdy.ar   = RESET;
      tbcrdy.clk  = PCICLK;

   "-------------------------------------------------------------------------------
   " DSP HPI Control Signals
   "-------------------------------------------------------------------------------
   "HPI control signals default values
      HCS_.ap    = RESET;       "HCS_ defaults to 1
      HDS1_.ap   = RESET;       "HDS1_ defaults to 1
      HRW.ar     = RESET;       "HRW defaults to 0

   "HPI control signals clocked by PCI clock
      HCS_.clk   = PCICLK;
      HDS1_.clk  = PCICLK;
      HRW.clk    = PCICLK;

   "HPI HCNTL[1..0] control signals
   "Decode host control bits based on target address
   "hcntl1 is a '1' for both BAR3 and BAR4 data register accesses
      hcntl1 =  (!PTATN_ & BAR3 & pt_addr1 & !pt_addr0) #
                (!PTATN_ & BAR4);

   "hcntl0 is a '1' for BAR3 address and data register accesses
      hcntl0 =  (!PTATN_ & BAR3 & pt_addr1) #
                (!PTATN_ & BAR3 & pt_addr0);

   "Host read/write is opposite polarity of PCI pass-thru read/write
      HRW    := !PTWR;

   "HPI ready synchronization
      hrdy     := !DSP_HRDY_;
      hrdy.ar  = RESET;
      hrdy.clk = PCICLK;

   "-------------------------------------------------------------------------------
   " PCI Add-on State Machine
   "-------------------------------------------------------------------------------
      aos.ar  = RESET;              "Default to AO_IDLE1 state at reset
      aos.clk = PCICLK;             "Add-on state machine clocked by PCI clock
      aos.ce  = !PCI_DET_;          "Add-on state machine only active in PCI slot

@dcstate

   "The add-on state machine drives the add-on control interface and
   "TBC/HPI control interface signals according to the states and tables
   "listed above.
```

```
state_diagram aos
"-------------------------------------------------------------------------------
" Wait for PCI or EMIF Access Request
"-------------------------------------------------------------------------------
"Loop waiting on PTATN_ to go active or EMIF PCI controller access request
state AO_IDLE1:
   ao_cntl := AOC_IDLE1;
   th_cntl := THC_IDLE1;

   "Check for EMIF request of PCI controller
   if (emif_req) then AO_IDLE2 with {
      ao_bsy := 0;
   }
   else
   "Check for PCI pass-thru access request
   if (!PTATN_ & !emif_req) then AO_IDLE3 with {
      ao_bsy := 1;

   }
   "If no EMIF or PCI access requests, state in this state
   else AO_IDLE1;

"EMIF request for PCI controller access
state AO_IDLE2:
   ao_cntl := AOC_IDLE2;
   th_cntl := THC_IDLE2;
   ao_bsy  := 0;
   emif_ack:= 0;

   "Determine type of EMIF PCI controller access (register read/write
   "or FIFO read/write)

   "Start the requested PT access.

   "EMIF PCI Controller Register Write Request
   if (!EA16 & !AWE_) then EMIF_WR1 with {
      ao_cntl := AOC_EMIF_WR1;
   }

   "EMIF PCI Controller Register Read Request
   else if (!EA16 & !ARE_) then EMIF_RD1 with {
      ao_cntl := AOC_EMIF_RD1;
   }

   "EMIF PCI FIFO Write Request
   else if (EA16 & !AWE_) then FIFO_WR1 with {
      ao_cntl := AOC_FIFO_WR1;
   }
```

```
   ″EMIF PCI FIFO Read Request
   else if (EA16 & !ARE_) then FIFO_RD1 with {
      ao_cntl := AOC_FIFO_RD1;
   }

   ″Ignore Invalid Requests
   else AO_IDLE1 with {
      ao_cntl := AOC_IDLE1;
   }

″-------------------------------------------------------------------------------
″ PCI Pass-thru Access Requested
″-------------------------------------------------------------------------------
″Pass-thru address is latched at end of this state in pt_addr[4..0]
state AO_IDLE3:
   th_cntl := THC_IDLE3;
   ao_bsy  := 1;
   emif_ack:= 0;

   ″Determine type of PCI pass-thru access (TBC,Register,HPI) and
   ″read or write.

   ″TBC Write
   if (TBC_WR) then TBC_WR1 with {
      ao_cntl := AOC_IDLE3;
   }

   ″TBC Read
   else if (TBC_RD) then TBC_RD1 with {
      ao_cntl := AOC_IDLE3;
   }

   ″PCI-mapped Register Write
   else if (REG_WR) then REG_WR1 with {
      ao_cntl := AOC_IDLE3;
   }

   ″PCI-mapped Register Read
   else if (REG_RD) then REG_RD1 with {
   ao_cntl := AOC_IDLE3;
   }

   ″HPI BAR3 Write
   else if (HPI_WR) then HPI_WR1 with {
      ao_cntl := AOC_IDLE3;
   }
```

```
    ″HPI BAR3 Read
    else if (HPI_RD) then HPI_RD1 with {
       ao_cntl := AOC_IDLE3;
    }

    ″HPI BAR4 Write
    else if (HPI_BURST_WR) then HPI_BURST_WR1 with {
       ao_cntl := AOC_IDLE4;
    }

    ″HPI BAR4 Read
    else if (HPI_BURST_RD) then HPI_BURST_RD1 with{
    ao_cntl := AOC_IDLE5;
    }

    else AO_IDLE1 with {
       ao_cntl := AOC_IDLE1;
    }

″-------------------------------------------------------------------------------
″ JTAG TBC Register Write
″-------------------------------------------------------------------------------
″Loop until TBC is ready.
state TBC_WR1:
   ao_cntl := AOC_TBC_WR1;
   th_cntl := THC_TBC_WR1;
   ao_bsy  := 1;
   emif_ack:= 0;

   ″Check TBC ready signal
   if (tbcrdy) then TBC_WR2
   else TBC_WR1;

state TBC_WR2:
   ao_cntl := AOC_TBC_WR2;
   th_cntl := THC_TBC_WR2;
   ao_bsy  := 1;
   emif_ack:= 0;
   goto TBC_WR3;

state TBC_WR3:
   ao_cntl := AOC_TBC_WR3;
   th_cntl := THC_TBC_WR3;
   ao_bsy  := 1;
   emif_ack:= 0;
   goto TBC_WR4;
```

```
"PCI controller data is strobed into TBC register
state TBC_WR4:
   ao_cntl := AOC_TBC_WR4;
   th_cntl := THC_TBC_WR4;
   ao_bsy  := 1;
   emif_ack:= 0;
   goto AO_DONE;

"-------------------------------------------------------------------------------
" JTAG TBC Register Read
"-------------------------------------------------------------------------------
"Loop until TBC is ready
state TBC_RD1:
   ao_cntl := AOC_TBC_RD1;
   th_cntl := THC_TBC_RD1;
   ao_bsy  := 1;
   emif_ack:= 0;

   if (tbcrdy) then TBC_RD2
   else TBC_RD1;

state TBC_RD2:
   ao_cntl := AOC_TBC_RD2;
   th_cntl := THC_TBC_RD2;
   ao_bsy  := 1;
   emif_ack:= 0;
   goto TBC_RD3;

state TBC_RD3:
   ao_cntl := AOC_TBC_RD3;
   th_cntl := THC_TBC_RD3;
   ao_bsy  := 1;
   emif_ack:= 0;
   goto TBC_RD4;

"TBC data is latched into PCI controller
state TBC_RD4:
   ao_cntl := AOC_TBC_RD4;
   th_cntl := THC_TBC_RD4;
   ao_bsy  := 1;
   emif_ack:= 0;
   goto AO_DONE;
```

```
"-------------------------------------------------------------------------------
" CPLD Register Write
"-------------------------------------------------------------------------------
state REG_WR1:
   ao_cntl := AOC_REG_WR1;
   th_cntl := THC_REG_WR1;
   ao_bsy  := 1;
   emif_ack:= 0;
   goto REG_WR2;

state REG_WR2:
   ao_cntl := AOC_REG_WR2;
   th_cntl := THC_REG_WR2;
   ao_bsy  := 1;
   emif_ack:= 0;
   goto REG_WR3;

"PCI controller data is latched into CPLD register
state REG_WR3:
   ao_cntl := AOC_REG_WR3;
   th_cntl := THC_REG_WR3;
   ao_bsy  := 1;
   emif_ack:= 0;
   goto AO_DONE;

"-------------------------------------------------------------------------------
" CPLD Register Read
"-------------------------------------------------------------------------------
state REG_RD1:
   ao_cntl := AOC_REG_RD1;
   th_cntl := THC_REG_RD1;
   ao_bsy  := 1;
   emif_ack:= 0;
   goto REG_RD2;

state REG_RD2:
   ao_cntl := AOC_REG_RD2;
   th_cntl := THC_REG_RD2;
   ao_bsy  := 1;
   emif_ack:= 0;
   goto REG_RD3;
```

```
state REG_RD3:
   ao_cntl := AOC_REG_RD3;
   th_cntl := THC_REG_RD3;
   ao_bsy  := 1;
   emif_ack:= 0;
   goto AO_DONE;

"-------------------------------------------------------------------------------
" DSP HPI Register Write
"-------------------------------------------------------------------------------
"Assert HCS- low and register HCNTLx bits based on target address
state HPI_WR1:
   ao_cntl := AOC_HPI_WR1;
   th_cntl := THC_HPI_WR1;
   ao_bsy  := 1;
   emif_ack:= 0;
   goto HPI_WR2;

"Assert HCS- low and setup HPI control signals
state HPI_WR2:
   ao_cntl := AOC_HPI_WR2;
   th_cntl := THC_HPI_WR2;
   ao_bsy  := 1;
   emif_ack:= 0;
   goto HPI_WR3;

"Assert HDS1- (HSTROBE-) low
state HPI_WR3:
   ao_cntl := AOC_HPI_WR3;
   th_cntl := THC_HPI_WR3;
   ao_bsy  := 1;
   emif_ack:= 0;
   goto HPI_WR4;

"Wait for HPI ready (after 1 clock synchronization)
"before bringing HDS1- high again
state HPI_WR4:
   ao_cntl := AOC_HPI_WR4;
   th_cntl := THC_HPI_WR4;
   ao_bsy  := 1;
   emif_ack:= 0;

   if (hrdy) then HPI_WR5;
   else HPI_WR4;
```

```
"Bring HDS1- (HSTROBE-) high to clock in 1st 16-bit word
state HPI_WR5:
   ao_cntl := AOC_HPI_WR5;
   th_cntl := THC_HPI_WR5;
   ao_bsy  := 1;
   emif_ack:= 0;
   goto HPI_WR6;


"Assert HDS1- (HSTROBE-) low to begin 2nd 16-bit word transfer
state HPI_WR6:
   ao_cntl := AOC_HPI_WR6;
   th_cntl := THC_HPI_WR6;
   ao_bsy  := 1;
   emif_ack:= 0;
   goto HPI_WR7;


"Bring HDS1- (HSTROBE-) high to clock in 2nd 16-bit word (triggers Aux DMA)
state HPI_WR7:
   ao_cntl := AOC_HPI_WR7;
   th_cntl := THC_HPI_WR7;
   ao_bsy  := 1;
   emif_ack:= 0;
   goto HPI_WR8;


"Bring HCS- high and assert PT_RDY_ low to PCI controller
state HPI_WR8:
   ao_cntl := AOC_HPI_WR8;
   th_cntl := THC_HPI_WR8;
   ao_bsy  := 1;
   emif_ack:= 0;
   goto AO_DONE;


"-------------------------------------------------------------------------------
" DSP HPI Register Read
"-------------------------------------------------------------------------------
"Assert HCS- low and register HCNTLx bits based on target address
state HPI_RD1:
   ao_cntl := AOC_HPI_RD1;
   th_cntl := THC_HPI_RD1;
   ao_bsy  := 1;
   emif_ack:= 0;
   goto HPI_RD2;
```

```
"Setup HPI control signals
state HPI_RD2:
   ao_cntl := AOC_HPI_RD2;
   th_cntl := THC_HPI_RD2;
   ao_bsy  := 1;
   emif_ack:= 0;
   goto HPI_RD3;

"Assert HDS1– (HSTROBE–) low to request read
state HPI_RD3:
   ao_cntl := AOC_HPI_RD3;
   th_cntl := THC_HPI_RD3;
   ao_bsy  := 1;
   emif_ack:= 0;
   goto HPI_RD4;

"Wait for data read and HPI ready clock synchronization
state HPI_RD4:
   ao_cntl := AOC_HPI_RD4;
   th_cntl := THC_HPI_RD4;
   ao_bsy  := 1;
   emif_ack:= 0;
   goto HPI_RD5;

"Wait for HPI ready (1st 16–bit data word available)
state HPI_RD5:
   ao_cntl := AOC_HPI_RD5;
   th_cntl := THC_HPI_RD5;
   ao_bsy  := 1;
   emif_ack:= 0;

   "When HPI is ready (data is available) goto next state
   if (hrdy) then HPI_RD6;
   else HPI_RD5;

"Set HDS1– high again, write 1st 16–bit data word into PCI controller
state HPI_RD6:
   ao_cntl := AOC_HPI_RD6;
   th_cntl := THC_HPI_RD6;
   ao_bsy  := 1;
   emif_ack:= 0;
   goto HPI_RD7;
```

```
"Assert HDS1- (HSTROBE-) low to request 2nd 16-bit data word
state HPI_RD7:
   ao_cntl := AOC_HPI_RD7;
   th_cntl := THC_HPI_RD7;
   ao_bsy  := 1;
   emif_ack:= 0;
   goto HPI_RD8;

"Set HDS1- high again, write 2nd 16-bit word into PCI controller
state HPI_RD8:
   ao_cntl := AOC_HPI_RD8;
   th_cntl := THC_HPI_RD8;
   ao_bsy  := 1;
   emif_ack:= 0;
   goto AO_DONE;

"-------------------------------------------------------------------------------
" DSP HPI Data Register Write (Address Auto-Inc Mode) with Burst Support
"-------------------------------------------------------------------------------
" Assert HCS- low and setup HPI control signals
state HPI_BURST_WR1:
   ao_cntl := AOC_HPI_BWR1;
   th_cntl := THC_HPI_BWR1;
   ao_bsy  := 1;
   emif_ack:= 0;
   goto HPI_BURST_WR2;

"Assert HDS1- (HSTROBE-) low
state HPI_BURST_WR2:
   ao_cntl := AOC_HPI_BWR2;
   th_cntl := THC_HPI_BWR2;
   ao_bsy  := 1;
   emif_ack:= 0;
   goto HPI_BURST_WR3;

"Wait for HPI ready (after 1 clock synchronization)
"before bring HDS1- high again
state HPI_BURST_WR3:
   ao_cntl := AOC_HPI_BWR3;
   th_cntl := THC_HPI_BWR3;
   ao_bsy  := 1;
   emif_ack:= 0;

   if (hrdy) then HPI_BURST_WR4;
   else HPI_BURST_WR3;
```

```
"Assert HDS1- (HSTROBE-) low to begin 2nd 16-bit word transfer
state HPI_BURST_WR4:
   ao_cntl := AOC_HPI_BWR4;
   th_cntl := THC_HPI_BWR4;
   ao_bsy  := 1;
   emif_ack:= 0;
   goto HPI_BURST_WR5;

"Bring HDS1- (HSTROBE-) high to clock in 2nd 16-bit word (triggers Aux DMA)
state HPI_BURST_WR5:
   ao_cntl := AOC_HPI_BWR5;
   th_cntl := THC_HPI_BWR5;
   ao_bsy  := 1;
   emif_ack:= 0;
   goto HPI_BURST_WR6;

"Bring HCS- high and assert PT_RDY_ low to PCI controller
state HPI_BURST_WR6:
   ao_cntl := AOC_HPI_BWR6;
   th_cntl := THC_HPI_BWR6;
   ao_bsy  := 1;
   emif_ack:= 0;
   goto HPI_BURST_WR7;

"Bring PT_RDY_ back high, wait for response from PCI controller
state HPI_BURST_WR7:
   ao_cntl := AOC_HPI_BWR7;
   th_cntl := THC_HPI_BWR7;
   ao_bsy  := 1;
   emif_ack:= 0;
   goto HPI_BURST_WR8;

"Check if another value is available from the PCI controller (burst)
state HPI_BURST_WR8:
   ao_cntl := AOC_HPI_BWR8;
   th_cntl := THC_HPI_BWR8;
   ao_bsy  := 1;
   emif_ack:= 0;

   case
     "PT is ready with another word.
     (!PTATN_ & !PTBURST_): HPI_BURST_WR1 with {}

     "Burst is over. Empty the pipe.
     (!PTATN_ &  PTBURST_): HPI_BURST_WR1 with {}

     "Burst is not over but PT is not ready with another word.
     ( PTATN_ & !PTBURST_): HPI_BURST_WR8 with {}
```

```
      ”Burst is over and the pipe is empty.
      ( PTATN_ &  PTBURST_): AO_DONE with {
        ao_cntl := AOC_DONE;
        th_cntl := THC_DONE;
      }
   endcase;

”-------------------------------------------------------------------------------
” DSP HPI Data Register Read (Address Auto–Inc Mode) with Burst Support
”-------------------------------------------------------------------------------
”Assert HCS– low and setup HPI control signals
state HPI_BURST_RD1:
   ao_cntl := AOC_HPI_BRD1;
   th_cntl := THC_HPI_BRD1;
   ao_bsy  := 1;
   emif_ack:= 0;
   goto HPI_BURST_RD2;

”Assert HDS1– (HSTROBE–) low to request read
state HPI_BURST_RD2:
   ao_cntl := AOC_HPI_BRD2;
   th_cntl := THC_HPI_BRD2;
   ao_bsy  := 1;
   emif_ack:= 0;
   goto HPI_BURST_RD3;

”Wait for data read and HPI ready clock synchronization
state HPI_BURST_RD3:
   ao_cntl := AOC_HPI_BRD3;
   th_cntl := THC_HPI_BRD3;
   ao_bsy  := 1;
   emif_ack:= 0;
   goto HPI_BURST_RD4;

”Wait for HPI ready (1st 16–bit data word available)
state HPI_BURST_RD4:
   ao_cntl := AOC_HPI_BRD4;
   th_cntl := THC_HPI_BRD4;
   ao_bsy  := 1;
   emif_ack:= 0;

   ”When HPI is ready (data is available) goto next state
   if (hrdy) then HPI_BURST_RD5;
   else HPI_BURST_RD4;
```

```
"Set HDS1– high again, write 1st 16–bit data word into PCI controller
state HPI_BURST_RD5:
   ao_cntl := AOC_HPI_BRD5;
   th_cntl := THC_HPI_BRD5;
   ao_bsy  := 1;
   emif_ack:= 0;

   if (hrdy) then HPI_BURST_RD6
   else HPI_BURST_RD5;

"Assert HDS1– (HSTROBE–) low to request 2nd 16–bit data word
state HPI_BURST_RD6:
   ao_cntl := AOC_HPI_BRD6;
   th_cntl := THC_HPI_BRD6;
   ao_bsy  := 1;
   emif_ack:= 0;
   goto HPI_BURST_RD7;

"Set HDS1– high again, write 2nd 16–bit word into PCI controller
"Assert PT_RDY_ low to end 32–bit word transfer
state HPI_BURST_RD7:
   ao_cntl := AOC_HPI_BRD7;
   th_cntl := THC_HPI_BRD7;
   ao_bsy  := 1;
   emif_ack:= 0;
   goto HPI_BURST_RD8;

"Bring HCS– and PT_RDY_ high, wait for response from PCI controller
state HPI_BURST_RD8:
   ao_cntl := AOC_HPI_BRD8;
   th_cntl := THC_HPI_BRD8;
   ao_bsy  := 1;
   emif_ack:= 0;
   goto HPI_BURST_RD9;

"Check if PCI controller wants another value (burst)
state HPI_BURST_RD9:
   ao_cntl := AOC_HPI_BRD9;
   th_cntl := THC_HPI_BRD9;
   ao_bsy  := 1;
   emif_ack:= 0;

   case
      "PT is ready with another word.
      (!PTATN_ & !PTBURST_): HPI_BURST_RD1 with {}

      "Burst is over. Empty the pipe.
      (!PTATN_ &  PTBURST_): HPI_BURST_RD1 with {}
```

```
        ”Burst is not over but PT is not ready with another word.
        ( PTATN_ & !PTBURST_): HPI_BURST_RD9 with {}

        ”Burst is over and the pipe is empty.
        ( PTATN_ &  PTBURST_): AO_DONE with {
          ao_cntl := AOC_DONE;
          th_cntl := THC_DONE;
        }
    endcase;

”-------------------------------------------------------------------------------
” EMIF PCI Controller Register Write
”-------------------------------------------------------------------------------
state EMIF_WR1:
    ao_cntl := AOC_EMIF_WR1;
    th_cntl := THC_EMIF_WR1;
    ao_bsy  := 0;
    emif_ack:= 1;
    goto EMIF_WR2;

state EMIF_WR2:
    ao_cntl := AOC_EMIF_WR2;
    th_cntl := THC_EMIF_WR2;
    ao_bsy  := 0;
    emif_ack:= 1;
    goto EMIF_WR3;

state EMIF_WR3:
    ao_bsy  := 0;
    ao_cntl := AOC_EMIF_WR3;
    th_cntl := THC_EMIF_WR3;
    emif_ack:= 1;

    if (AWE_) then AO_IDLE1 with {
        ao_cntl := AOC_DONE;
        th_cntl := THC_DONE;
        emif_ack:= 0;
    }
    else EMIF_WR3;
```

```
"-------------------------------------------------------------------------------
" EMIF PCI Controller Register Read
"-------------------------------------------------------------------------------
state EMIF_RD1:
   ao_cntl := AOC_EMIF_RD1;
   th_cntl := THC_EMIF_RD1;
   ao_bsy  := 0;
   emif_ack:= 1;
   goto EMIF_RD2;

state EMIF_RD2:
   ao_bsy  := 0;
   ao_cntl := AOC_EMIF_RD2;
   th_cntl := THC_EMIF_RD2;
   emif_ack:= 1;

   if (ARE_) then AO_IDLE1 with {
      ao_cntl := AOC_DONE;
      th_cntl := THC_DONE;
      emif_ack:= 0;
   }
   else EMIF_RD2;

"-------------------------------------------------------------------------------
" EMIF PCI Controller FIFO Write
"-------------------------------------------------------------------------------
state FIFO_WR1:
   ao_cntl := AOC_FIFO_WR1;
   th_cntl := THC_FIFO_WR1;
   ao_bsy  := 0;
   emif_ack:= 1;
   goto FIFO_WR2;

state FIFO_WR2:
   ao_bsy  := 0;
   emif_ack:= 1;
   ao_cntl := AOC_FIFO_WR2;
   th_cntl := THC_FIFO_WR2;

   if (AWE_) then AO_IDLE1 with  {
      ao_cntl := AOC_DONE;
      th_cntl := THC_DONE;
      emif_ack:= 0;
   }
   else FIFO_WR2;
```

```
    ”-------------------------------------------------------------------------------
    ” EMIF PCI Controller FIFO Read
    ”-------------------------------------------------------------------------------
    state FIFO_RD1:
       ao_cntl := AOC_FIFO_RD1;
       th_cntl := THC_FIFO_RD1;
       ao_bsy  := 0;
       emif_ack:= 1;
       goto FIFO_RD2;

    state FIFO_RD2:
       ao_bsy  := 0;
       emif_ack:= 1;
       ao_cntl := AOC_FIFO_RD2;
       th_cntl := THC_FIFO_RD2;

       if (ARE_) then FIFO_RD3 with {
          ao_cntl := AOC_FIFO_RD3;
          emif_ack:= 0;
       }
       else FIFO_RD2;

    state FIFO_RD3:
       ao_cntl := AOC_FIFO_RD3;
       th_cntl := THC_FIFO_RD3;
       ao_bsy  := 0;
       emif_ack:= 0;
       goto AO_IDLE1;

    ”-------------------------------------------------------------------------------
    ” Add-on Access Done
    ”-------------------------------------------------------------------------------
    state AO_DONE:
       ao_cntl := AOC_DONE;
       th_cntl := THC_DONE;
       ao_bsy  := 0;
       emif_ack:= 0;
       goto AO_IDLE1;

END
```

Appendix D

TMS320C6x EVM PCI Configuration EEPROM

This appendix documents the EEPROM used on the ’C6x EVM 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 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 EVM 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	0x1002	EVM
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 ’C6x 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).

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 'C6x 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*.

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.

H

halfword: The 'C6x 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 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.

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.

light emitting diode (LED): A semiconductor chip that gives off visible or infrared light when activated.

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*.

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.

µF: 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).

McBSP: See *multichannel buffered serial port.*

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 (μF): 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*.

μs: See *microsecond*.

ms: See *millisecond*.

MSbyte: See *most significant byte*.

multichannel buffered serial port (McBSP): A standard serial port interface found on 'C6x 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.)

mutex: A mutual exclusion semaphore used to restrict access to a resource.

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 memory*.

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

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.

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.

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.

V

V: See *volt.*

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*.

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 'C6x 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.

Index

A

B

C

D

E

F

G

H

N

O

P

R

S

T

U

V

W