

# ***TMS320C8x Software Development Board Programmer's Guide***

Literature Number: SPRU178  
Manufacturing Part Number: D418019-9761 revision \*  
January 1997



Printed on Recycled Paper

## **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 programmer's guide for the TMS320C8x ('C8x) software development board (SDB) provides application programming interface (API) references, descriptions of hardware functions of the SDB, theory of operation, and example code to help you develop custom applications with the SDB.

This manual assumes you are familiar with working in a Windows NT™ environment and understand general and technical PC™ and multimedia processes and terminology.

### ***How to Use This Manual***

This book is divided into three distinct sections:

- ❑ **Introductory information**, consisting of Chapters 1 and 2. Chapter 1 provides an overview of the TMS320C8x SDB, its components, and the organization of this book. Chapter 2 discusses the theory of hardware operation of the SDB.
- ❑ **Topical material**, consisting of Chapters 3–6, provides descriptions of hardware functions and a complete API reference for each of the following topics:
  - Audio capture and playback
  - Video display
  - Video capture
  - Host communications
- ❑ **Reference material**, consisting of Appendixes A, B, C, and D, provides example code, a listing of the shared data types and macros, a reference listing of all API functions, and a glossary.

## Notational Conventions

This document uses the following conventions.

- Program listings, program examples, and interactive displays are shown in a special typeface. Examples use a **bold version** of the special typeface for emphasis, such as in the following listing:

```
long DisplaySemaId;
ULONG Buff;
DisplaySemaId = TaskOpenSema(-1,0);
Display_Init();
Display_InstallSema(DisplaySemaId);
Display_SetMode(640,480,60,DISPLAY_T555,DISPLAY_VIDEO);
Display_Enable();
while (1) {
    Display_ToggleBuffers();
    TaskWaitSema(DisplaySemaId);
    Buff = Display_GetBuffer(DISPLAY_INACTIVE);
    /* do some processing here */
}
```

- Device pins and register bits often are represented in groups. Different notation distinguishes between device pins and register bits.
  - 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, D[63:0] represents the 64 data pins (D63 through D0) on a device.
  - Register bit group notation consists of the register name or the bit field name followed by parentheses containing the range of bits included in the group. A colon separates the numbers in the range. For example, CDCIDX(7:0) represents the eight bits of the audio codec's index address register, and IXA(4:0) represents the five IXA bits (IXA4 through IXA0) of the CDCIDX register.

## Information About Cautions

This book contains cautions.

**This is an example of a caution statement.**

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

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

## **Related Documentation From Texas Instruments**

The following books describe the TMS320C8x software development board 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.

***TMS320C8x Software Development Board Installation Guide*** (literature number SPRU150B) provides information about how to install and use the SDB.

***TMS320C80 Digital Signal Processor Data Sheet*** (literature number SPRS023) describes the features of the TMS320C80 and provides pinouts, electrical specifications, and timings for the device.

***TMS320C80 (MVP) C Source Debugger User's Guide*** (literature number SPRU107) describes the 'C8x master processor and parallel processor C source debuggers. This manual provides information about the features and operation of the debuggers and the parallel debug manager; it also includes basic information about C expressions and a description of progress and error messages.

***TMS320C80 (MVP) Code Generation Tools User's Guide*** (literature number SPRU108) provides information about the features and operation of the linker and the master processor (MP) and parallel processor (PP) C compilers and assemblers. It also includes a description of the common object file format (COFF) and shows you how to link MP and PP code.

***TMS320C8x Master Processor User's Guide*** (literature number SPRU109) provides information about the master processor (MP) features, architecture, operation, and assembly language instruction set; it also includes sample applications that illustrate various MP operations.

***TMS320C8x Multitasking Executive User's Guide*** (literature number SPRU112) provides information about the multitasking executive software features, operation, and interprocessor communications; it also includes a list of task error codes.

***TMS320C8x Parallel Processor User's Guide*** (literature number SPRU110) provides information about the parallel processor (PP) features, architecture, operation, and assembly language instruction set; it also includes software applications and optimizations.

***TMS320C8x System-Level Synopsis*** (literature number SPRU113) describes the 'C8x features, development environment, architecture, memory organization, and communication network (the crossbar).

**TMS320C80 Transfer Controller User's Guide** (literature number SPRU105) provides information about the transfer controller (TC) features, functional blocks, and operation; it also includes examples of block write operations for big- and little-endian modes.

**TMS320C80 Video Controller User's Guide** (literature number SPRU111) provides information about the video controller (VC) features, architecture, and operation; it also includes procedures and examples for programming the serial register transfer (SRT) controller and the frame timer registers.

**TVP3020 Video Interface Palette Data Manual** (literature number SLAS080) provides information about the TVP3020 video interface palette features, register set, operation, and characteristics.

### ***FCC Warning***

This equipment is intended for use in a laboratory test environment only. It generates, uses, and can radiate radio frequency energy and has not been tested for compliance with the limits of computing devices pursuant to subpart J of part 15 of FCC rules, which are designed to provide reasonable protection against radio frequency interference. Operation of this equipment in other environments may cause interference with radio communications, in which case the user at his own expense will be required to take whatever measures may be required to correct this interference.

### ***Trademarks***

IBM and PC are trademarks of International Business Machines Corporation.

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

**If You Need Assistance. . .**

|  |  |   |                    |
|--|--|---|--------------------|
| <input type="checkbox"/> <b>World-Wide Web Sites</b>   |  |   |                    |
| TI Online  |  | <a href="http://www.ti.com">http://www.ti.com</a>   |                    |
| Semiconductor Product Information Center (PIC)   |  | <a href="http://www.ti.com/sc/docs/pic/home.htm">http://www.ti.com/sc/docs/pic/home.htm</a>           |                    |
| DSP Solutions  |  | <a href="http://www.ti.com/dsps">http://www.ti.com/dsps</a>   |                    |
| 320 Hotline On-line™   |  | <a href="http://www.ti.com/sc/docs/dsps/support.html">http://www.ti.com/sc/docs/dsps/support.html</a> |                    |
| <input type="checkbox"/> <b>North America, South America, Central America</b>  |  |   |                    |
| 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 <a href="ftp://ftp.ti.com/mirrors/tms320bbs">ftp://ftp.ti.com/mirrors/tms320bbs</a>  |  |   |                    |
| <input type="checkbox"/> <b>Europe, Middle East, Africa</b>  |  |   |                    |
| 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   |                    |
| <input type="checkbox"/> <b>Asia-Pacific</b>   |  |   |                    |
| 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 <a href="ftp://dsp.ee.tit.edu.tw/pub/TI/">ftp://dsp.ee.tit.edu.tw/pub/TI/</a>   |  |   |                    |
| <input type="checkbox"/> <b>Japan</b>  |  |   |                    |
| 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"                        |   |                    |
| <input type="checkbox"/> <b>Documentation</b>  |  |   |                    |
| 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   |  | Email: <a href="mailto:comments@books.sc.ti.com">comments@books.sc.ti.com</a>                         |                    |
| Technical Documentation Services, MS 702   |  |   |                    |
| P.O. Box 1443  |  |   |                    |
| Houston, Texas 77251-1443  |  |   |                    |

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

# Contents

---

---

---

|          |  |            |
|----------|--|------------|
| <b>1</b> | <b>Introduction</b>  | <b>1-1</b> |
|          | <i>Describes the TMS320C8x software development board (SDB) and introduces you to the topics of this book.</i> |            |
| 1.1      | TMS320C8x Digital Signal Processor   | 1-2        |
| 1.2      | TMS320C8x SDB Contents and Components  | 1-2        |
| 1.3      | TMS320C8x SDB Hardware Functions   | 1-4        |
| 1.4      | TMS320C8x SDB Peripheral Driver Libraries  | 1-4        |
| <b>2</b> | <b>SDB Hardware</b>  | <b>2-1</b> |
|          | <i>Discusses the theory of hardware operation for the SDB.</i>   |            |
| 2.1      | System-Level Overview  | 2-2        |
| 2.1.1    | SDB Hardware Modules   | 2-2        |
| 2.1.2    | Data Buses   | 2-5        |
| 2.2      | TMS320C80  | 2-6        |
| 2.3      | Audio  | 2-7        |
| 2.3.1    | Memory-Mapped Audio Registers  | 2-8        |
| 2.3.2    | Audio Codec  | 2-13       |
| 2.3.3    | IDT72520 Bidirectional Bus-Matching FIFO   | 2-14       |
| 2.4      | Video Display  | 2-19       |
| 2.4.1    | Memory-Mapped Video Display Registers  | 2-21       |
| 2.4.2    | ICS1574 Pixel Clock Generator  | 2-30       |
| 2.4.3    | Supported Resolutions  | 2-31       |
| 2.4.4    | Video Overlay Feature (Mixed Mode)   | 2-33       |
| 2.5      | Video Capture  | 2-35       |
| 2.5.1    | Video Capture FIFO   | 2-35       |
| 2.5.2    | Memory-Mapped Video Capture Registers  | 2-37       |
| 2.5.3    | SAA7196 Video Decoder/Scaler (DESC)  | 2-44       |
| 2.5.4    | PCF8584 I <sup>2</sup> C Bus Controller  | 2-44       |
| 2.6      | Memory Controller  | 2-45       |
| 2.6.1    | DRAM   | 2-45       |
| 2.6.2    | VRAM   | 2-46       |
| 2.6.3    | I/O Bus  | 2-46       |
| 2.6.4    | Video Capture FIFO   | 2-46       |
| 2.6.5    | PCI Bus  | 2-47       |



|          |  |            |
|----------|--|------------|
| 2.7      | Interrupt Controller .....   | 2-48       |
| 2.7.1    | Memory-Mapped Interrupt Controller Registers .....   | 2-51       |
| 2.8      | PCI Interface .....  | 2-58       |
| 2.8.1    | PCI Status Register .....  | 2-59       |
| 2.8.2    | PCI FIFO .....   | 2-62       |
| 2.8.3    | TMS320C80 Access to the PCI FIFO .....   | 2-64       |
| 2.8.4    | PCI Plug and Play .....  | 2-64       |
| 2.8.5    | Host Address Space .....   | 2-66       |
| 2.8.6    | PCI Bus Mastering .....  | 2-68       |
| 2.8.7    | Bootstrapping .....  | 2-70       |
| <b>3</b> | <b>Audio Capture and Playback API .....</b>  | <b>3-1</b> |
|          | <i>Discusses the shared data types and macros and lists alphabetically the API functions associated with the audio capture and playback drivers for the SDB.</i> |            |
| 3.1      | Audio Capture and Playback API Macros and Data Types .....   | 3-2        |
| 3.1.1    | AUDIO_PTR Data Type .....  | 3-4        |
| 3.1.2    | Audio Metric Parameters .....  | 3-4        |
| 3.2      | Audio Buffering .....  | 3-5        |
| 3.2.1    | Individual Queue .....   | 3-7        |
| 3.2.2    | Individual Buffer .....  | 3-8        |
| 3.2.3    | Individual Subblock .....  | 3-8        |
| 3.3      | Audio Capture and Playback API Functions .....   | 3-9        |
| <b>4</b> | <b>Video Display API .....</b>   | <b>4-1</b> |
|          | <i>Discusses the video display data types and macros and lists alphabetically the API functions associated with the video display driver for the SDB.</i>        |            |
| 4.1      | Video Display API Macros and Data Types .....  | 4-2        |
| 4.1.1    | Color Modes .....  | 4-5        |
| 4.1.2    | Monitor Timing Parameters .....  | 4-5        |
| 4.1.3    | Metric Parameters .....  | 4-8        |
| 4.2      | Video Overlay .....  | 4-8        |
| 4.3      | Video Display Window .....   | 4-9        |
| 4.4      | Video Display API Functions .....  | 4-10       |
| <b>5</b> | <b>Video Capture API .....</b>   | <b>5-1</b> |
|          | <i>Discusses the data types and macros and lists alphabetically the API functions associated with the video capture driver for the SDB.</i>                      |            |
| 5.1      | Video Capture API Macros and Data Types .....  | 5-2        |
| 5.1.1    | Supported Scaling Resolutions .....  | 5-4        |
| 5.1.2    | Video Capture Metric Parameters .....  | 5-4        |
| 5.2      | Video Capture Buffering .....  | 5-5        |
| 5.3      | Video Capture API Functions .....  | 5-8        |

|          |   |            |
|----------|---|------------|
| <b>6</b> | <b>Host Communications API</b>  | <b>6-1</b> |
|          | <i>Discusses the data types and macros and lists alphabetically the API functions associated with the host communications driver for the SDB.</i>   |            |
| 6.1      | Host Communications API Macros and Data Types   | 6-2        |
| 6.2      | Interaction Between the Client API and the Server API   | 6-6        |
| 6.2.1    | Client/Server Synchronization   | 6-6        |
| 6.2.2    | Client to Server Protocol   | 6-8        |
| 6.3      | Bootstrapping the SDB from the Host   | 6-10       |
| 6.4      | Host Communications API Functions   | 6-11       |
| <b>A</b> | <b>Example Code</b>   | <b>A-1</b> |
|          | <i>Lists example code to illustrate the effective use of the API libraries. Some examples may be good starting points for developing your own applications.</i>   |            |
| A.1      | Video Capture-Process-Display Example   | A-2        |
| A.2      | Audio DMA Capture Example   | A-8        |
| A.3      | Audio DMA Playback Example  | A-12       |
| A.4      | Audio Block Capture/Playback Example  | A-17       |
| A.5      | Audio Programmed I/O Example  | A-21       |
| A.6      | Video Capture Scaling Example   | A-24       |
| A.7      | Video Display Test Example  | A-30       |
| <b>B</b> | <b>Shared Data Types and Macros</b>   | <b>B-1</b> |
|          | <i>Describes the data types and macros that are shared by the audio capture and playback, video display, video capture, and host communications drivers for the SDB.</i>                                |            |
| B.1      | TMS320C80 API Library Header File <sdbdrvs.h>   | B-2        |
| B.2      | Host API Library Header File <hsdbdrvs.h>   | B-4        |
| <b>C</b> | <b>API Functions With Arguments and Return Types</b>  | <b>C-1</b> |
|          | <i>Lists all the API functions in alphabetical order by function name and includes their arguments and return types. Use this list as a function protocol list and as a reminder of argument order.</i> |            |
| <b>D</b> | <b>Glossary</b>   | <b>D-1</b> |
|          | <i>Defines acronyms and key terms used in this book.</i>  |            |

# Figures

|       |   |      |
|-------|---|------|
| 1-1.  | TMS320C8x SDB Components .....  | 1-3  |
| 2-1.  | System Block Diagram .....  | 2-3  |
| 2-2.  | Audio Block Diagram .....   | 2-7  |
| 2-3.  | Audio FIFO Block Diagram .....  | 2-9  |
| 2-4.  | Video Display Block Diagram .....   | 2-20 |
| 2-5.  | Display EPLD Block Diagram .....  | 2-27 |
| 2-6.  | Display I/O Block Diagram .....   | 2-34 |
| 2-7.  | Video Capture Block Diagram .....   | 2-36 |
| 2-8.  | Video Digitizer Block Diagram .....   | 2-39 |
| 2-9.  | PCI Interface Block Diagram .....   | 2-58 |
| 2-10. | PCI FIFO Block Diagram .....  | 2-63 |
| 2-11. | Host Address Space .....  | 2-66 |
| 3-1.  | Audio API Data Types .....  | 3-3  |
| 3-2.  | Internal Audio Buffer Structure .....                                       | 3-5  |
| 3-3.  | Buffering Queue Structure .....   | 3-6  |
| 4-1.  | Video Display API Data Types .....  | 4-4  |
| 4-2.  | Relationship Between the Display Resolution and the Video Frame .....       | 4-5  |
| 4-3.  | Horizontal Sync and Porch Times of the Video Frame .....                    | 4-6  |
| 4-4.  | Video Display Driver Window Feature .....                                   | 4-9  |
| 5-1.  | Video Capture API Data Type (Metrics Parameter Structure CAPTURE_MET) ..... | 5-3  |
| 5-2.  | Video Capture Double Buffering Logic .....                                  | 5-7  |
| 6-1.  | Host Communications API Data Type (CLIENT_STAT) .....                       | 6-5  |
| 6-2.  | Client/Server Synchronization .....   | 6-7  |
| 6-3.  | Client/Server Command Flow .....  | 6-9  |

# Tables

|       |   |      |
|-------|---|------|
| 2-1.  | Audio Registers Summary .....                                 | 2-8  |
| 2-2.  | Commands Supported by the IDT72520 .....                      | 2-15 |
| 2-3.  | IDT72520 Internal Configuration Registers .....               | 2-16 |
| 2-4.  | FIFO Flag Pin Configurations .....                            | 2-17 |
| 2-5.  | Truth Table for FIFO Flag Assignments .....                   | 2-18 |
| 2-6.  | Video Display Registers Summary .....                         | 2-21 |
| 2-7.  | Standard Resolutions Supported by the API .....               | 2-32 |
| 2-8.  | Video Capture Registers Summary .....                         | 2-37 |
| 2-9.  | Category-1 Events .....                                       | 2-49 |
| 2-10. | Category-2 Events .....                                       | 2-50 |
| 2-11. | Interrupt Controller Registers Summary .....                  | 2-51 |
| 2-12. | Parts of the FIFO Device Accessed by Host/SDB Transfers ..... | 2-63 |
| 2-13. | PCI Bus Mastering Registers Summary .....                     | 2-69 |
| 3-1.  | Audio API Macros .....  | 3-2  |
| 4-1.  | Video Display API Macros .....                                | 4-2  |
| 5-1.  | Video Capture API Macros .....                                | 5-2  |
| 6-1.  | Host Communications API Macros .....                          | 6-2  |

# Examples

---

---

---

|       |  |      |
|-------|--|------|
| 2-1.  | Sample C Code to Reset the Audio FIFO .....                              | 2-15 |
| 2-2.  | Accessing Internal Configuration Register 4 .....                        | 2-16 |
| 2-3.  | Write Three RGB Triples to the Color Palette RAM .....                   | 2-22 |
| 2-4.  | Read Three RGB Triples from the Color Palette RAM .....                  | 2-23 |
| 2-5.  | Usage of TVPIDX and TVPDAT Registers .....                               | 2-24 |
| 2-6.  | Sample Code for Programming the ICS1574 .....                            | 2-30 |
| 2-7.  | Programming CAP to Be Triggered by an Odd-to-Even Field Transition ..... | 2-42 |
| 2-8.  | Setting the CAP Enable Bits .....  | 2-52 |
| 2-9.  | Setting/Clearing Category-2 Events .....                                 | 2-53 |
| 2-10. | Clearing an Event .....  | 2-56 |
| 4-1.  | Sample MT Table of Custom Timing Parameters .....                        | 4-8  |
| A-1.  | video .....  | A-2  |
| A-2.  | audcapt .....  | A-8  |
| A-3.  | audplay .....  | A-12 |
| A-4.  | audtest .....  | A-17 |
| A-5.  | piotest .....  | A-21 |
| A-6.  | capptest .....   | A-24 |
| A-7.  | disptest .....   | A-30 |

# Introduction

---

---

---

The software development board (SDB) is a peripheral component interconnect (PCI) plug-in card. The SDB helps you evaluate characteristics of the TMS320C8x digital signal processor (DSP) to determine how it will meet the requirements of your given application. You can also use the SDB as a development tool to create software applications for the 'C8x on a PC. The SDB is designed for use on PCI PC-based computers with Windows NT.

This chapter briefly describes the items that are delivered as part of the SDB and introduces you to the topics of this book.

| Topic   | Page |
|---|------|
| 1.1 TMS320C8x Digital Signal Processor .....        | 1-2  |
| 1.2 TMS320C8x SDB Contents and Components .....     | 1.2  |
| 1.3 TMS320C8x SDB Hardware Functions .....          | 1-4  |
| 1.4 TMS320C8x SDB Peripheral Driver Libraries ..... | 1-4  |

## 1.1 TMS320C8x Digital Signal Processor

The 'C8x is TI's first generation of single-chip multiprocessor DSP devices. A single 'C8x contains up to five powerful, fully programmable processors: a master processor (MP) and up to four parallel processors (PPs). The MP is a 32-bit reduced instruction set computer (RISC) processor with an integral, high-performance IEEE-754 floating-point unit. Each PP is a 64-bit advanced DSP that combines capabilities similar to those of conventional DSPs with advanced features to accelerate operation on a variety of data types.

The 'C8x supports a variety of parallel-processing configurations, which facilitate a wide range of DSP applications that require high processing speeds. Applications include medical and industrial image processing, three-dimensional graphics, virtual reality, digital audio and video compression, and telecommunications.

## 1.2 TMS320C8x SDB Contents and Components

The 'C8x SDB kit contains the following items:

- ☐ 'C8x SDB PCI plug-in card
- ☐ Three peripheral cables
- ☐ 'C8x SDB system software
- ☐ 'C8x C source debugger software
- ☐ User documentation

The 'C8x PCI plug-in card is the main component of the 'C8x SDB. The board is a printed-circuit assembly (PCA) that plugs into a PCI expansion slot on your computer's motherboard.

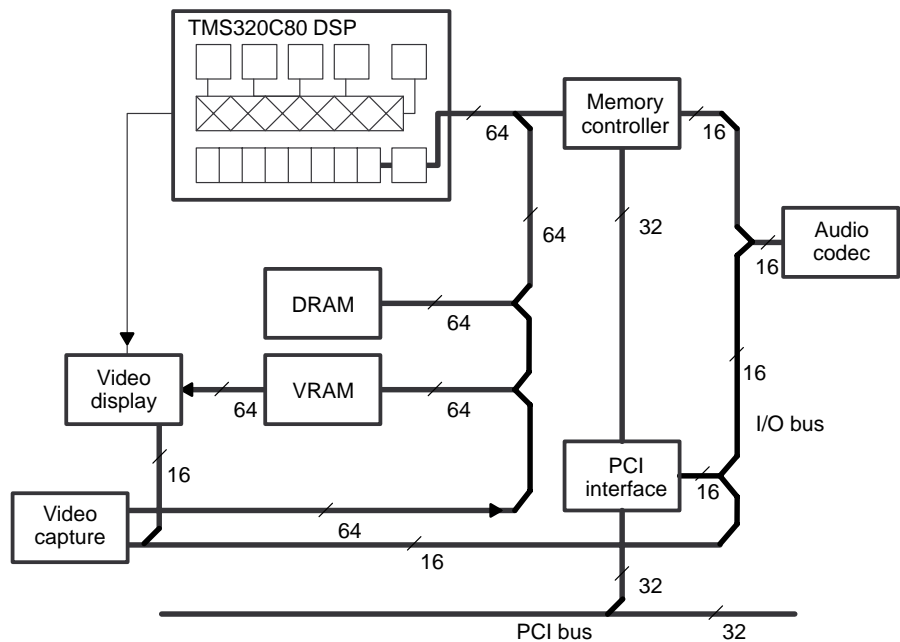
The 'C8x SDB PCI plug-in card includes the following components (see Figure 1-1):

- ☐ 40-MHz TMS320C80 DSP
- ☐ 8M bytes of dynamic random-access memory (DRAM)
- ☐ 2M bytes of video random-access memory (VRAM) for high-resolution display
- ☐ Audio codec for the capture and playback of audio signals at sample rates of up to 48 kHz in 16-bit stereo
- ☐ Video capture circuitry, consisting of a complete video front end for capturing National Television Standards Committee (NTSC) or phase alternation line (PAL) video signals in a composite video (CVBS) or super VHS (S-VHS) component format

- ❑ Video display circuitry, consisting of a video interface palette (VIP) that drives monitor resolutions up to  $1600 \times 1200$  at 8 BPP (bits per pixel) with a 60-Hz refresh rate
- ❑ PCI interface
- ❑ Memory controller

The SDB card also contains IEEE 1149.1-standard emulation support, and the card's retaining bracket has cable connectors that connect optional peripherals to the SDB.

Figure 1–1. TMS320C8x SDB Components





## 1.3 TMS320C8x SDB Hardware Functions

The 'C8x SDB is a complete 'C8x development platform. The board contains the following hardware functions:

|                                   |  |
|-----------------------------------|--|
| <b>Audio capture and playback</b> | The board contains hardware for the capture and playback of audio signals at sample rates of up to 48 kHz in 16-bit stereo.  |
| <b>Video display</b>              | A video RAMDAC coupled with 2M bytes of VRAM gives the board complete video display capabilities at resolutions of up to 1600 × 1200 pixels at 8 BPP (bits per pixel). |
| <b>Video capture</b>              | On a daughter card mounted on the main board is a complete video front end for capturing NTSC or PAL video signals in either S-VHS or CVBS component form.             |
| <b>Host communications</b>        | Also on board is in-circuit emulation hardware controlled by C source debuggers. These debuggers allow real-time, multiple-processor debugging of 'C8x code.           |

## 1.4 TMS320C8x SDB Peripheral Driver Libraries

The SDB gives you a platform for 'C8x performance evaluation, code benchmarking, and code production prior to building your own system. The ability to develop and successfully run code on a 'C8x during the early stages of application design greatly reduces your time to market. Therefore, it is not ideal to program all of the peripherals on the board before working on applications.

TI provides a set of 'C8x libraries that you can call through application programming interface (API) functions to set up the various hardware peripherals on the SDB. Using this set of libraries, you can start writing application code without having to program the hardware. However, TI understands that knowledge of the low-level activities happening on the board is useful if not necessary. Therefore, TI supplies all source code to these libraries for reference.

# SDB Hardware

---

---

---

This chapter discusses the theory of operation for the TMS320C8x software development board (SDB).

| Topic                           | Page |
|---------------------------------|------|
| 2.1 System-Level Overview ..... | 2-2  |
| 2.2 TMS320C80 .....             | 2-6  |
| 2.3 Audio .....                 | 2-7  |
| 2.4 Video Display .....         | 2-19 |
| 2.5 Video Capture .....         | 2-35 |
| 2.6 Memory Controller .....     | 2-45 |
| 2.7 Interrupt Controller .....  | 2-48 |
| 2.8 PCI Interface .....         | 2-58 |

## 2.1 System-Level Overview

Looking at the 'C8x SDB as a whole, it consists of the eight modules depicted in Figure 2–1. Each of these modules is discussed individually later in the chapter. This section describes the overall features of the hardware modules and the interaction between them.

**Note:**

The SDB operates in big-endian mode.

### 2.1.1 SDB Hardware Modules

The 'C8x SDB contains the following hardware modules:

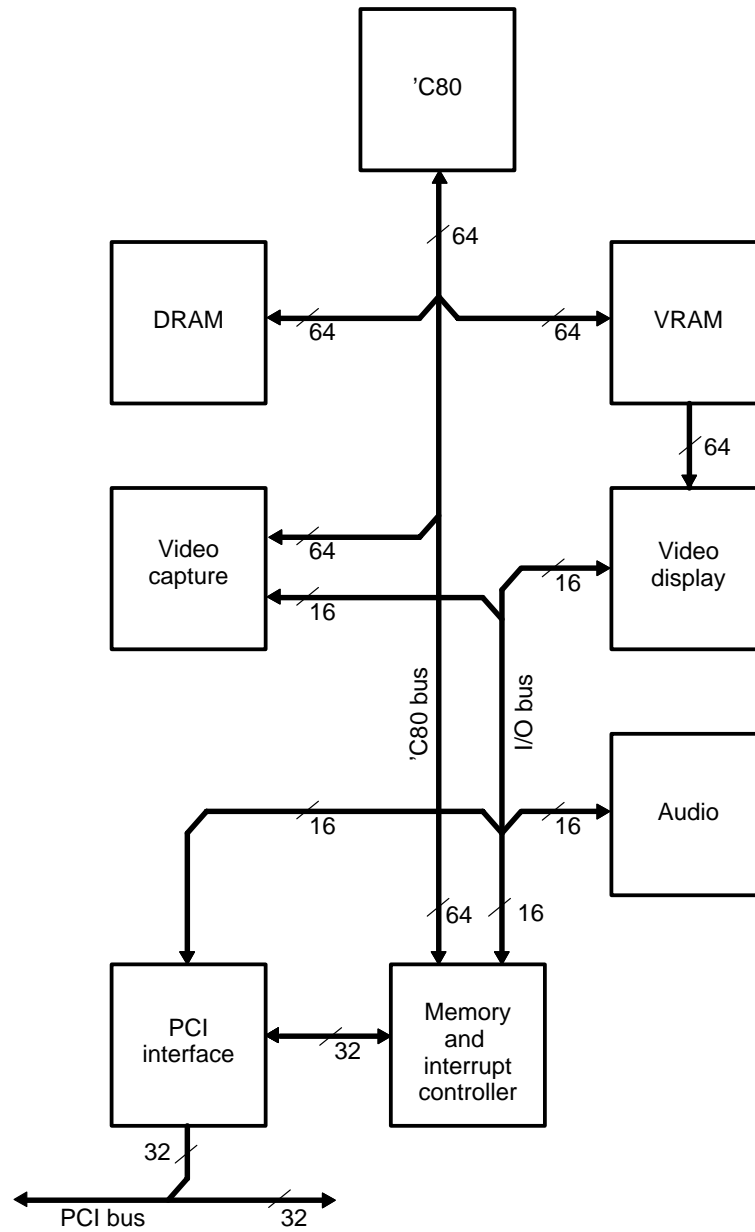
- ❑ **'C80:** TI's TMS320C80 digital signal processor (DSP) is the single-chip, multiprocessor device responsible for program execution and input/output management. For more information about the 'C80 multiprocessor DSP, see Section 2.2, *TMS320C80*.
- ❑ **Audio:** The on-board hardware for sampling and playing back audio data features:
  - Codec
  - Stereo
  - Sampling rate of up to 48 kHz
  - FIFO (first in, first out logic) interface

For more information about the audio hardware, see Section 2.3, *Audio*.

- ❑ **Video display:** The on-board video display hardware features:
  - TI's TVP3020 random-access memory digital-to-analog converter (RAMDAC)
  - Resolution support from 640 × 480 at 32 BPP to 1600 × 1200 at 8 BPP
  - Programmable pixel clock generator
  - Standard 15-pin D-sub RGB (red-green-blue) output
  - Multiplexing capability with analog VGA input

For more information about the video display hardware, see Section 2.4, *Video Display*.

Figure 2–1. System Block Diagram



- ❑ **Video capture:** The video capture hardware is a complete video front end capable of capturing video signals in either S-VHS or CVBS formats. The video capture front end includes:

- Two high-speed analog-to-digital converters (S-VHS or CVBS inputs)
- Video decoder accepting NTSC or PAL input formats
- Video scaler with independent horizontal and vertical scaling
- FIFO interface

For more information about the video capture hardware, see Section 2.5, *Video Capture*.

- ❑ **Memory/interrupt controller:** The memory/interrupt controller manages external memory plus advanced event handling. Remember, the SDB operates in big-endian mode only. This hardware is responsible for:

- Interfacing between buses ('C80 bus, I/O bus, and PCI bus)
- Managing special bus cycles
- Managing all board events (interrupts)

For more information about the memory/interrupt controller, see Section 2.6, *Memory Controller*, and Section 2.7, *Interrupt Controller*.

- ❑ **PCI interface:** A dual-ported FIFO interfaces the SDB with the host. The peripheral component interconnect (PCI) interface manages SDB transfers on the PCI bus. For more information about the PCI interface, see Section 2.8, *PCI Interface*.

- ❑ **DRAM:** The DRAM, used for program and data storage, is:

- 8M bytes total
- 64 bits wide
- Byte addressable
- Accessed with 3-cycle-per-column reads and 2-cycle-per-column writes

For more information about the DRAM, see subsection 2.6.1, *DRAM*.

- ❑ **VRAM:** The VRAM, used for video frame storage, is:

- 2M bytes total
- 64 bits wide
- Byte addressable
- Accessed with 3-cycle-per-column reads and 2-cycle-per-column writes

For more information about the VRAM, see subsection 2.6.2, *VRAM*.

### 2.1.2 Data Buses

Figure 2–1 shows the data paths between the hardware modules on the 'C8x SDB. There are three main data buses:

- ☐ 'C80 bus
- ☐ Input/output (I/O) bus
- ☐ PCI bus

The *'C80 bus* is the 64-bit-wide data bus of the 'C80. All transfers to and from the 'C80 happen on this bus. The 'C80 bus interfaces to the system DRAM, VRAM, and the video capture FIFO. The 'C80 bus also is used by the memory/interrupt controller to route data to and from the other two buses.

The *I/O bus* is designed as a general-purpose peripheral bus. In fact, all peripherals on the board interface to this bus to give access to internal peripheral registers. The peripherals interfaced to the I/O bus include the audio codec, display RAMDAC, video capture chipset, and register sets in the electrically programmable logic devices (EPLDs).

The *PCI bus* is an integral part of the host in that all transfers to and from the host happen over this bus. The SDB occupies a certain address range in the PCI address space as dictated by the PCI BIOS.

## 2.2 TMS320C80

The 'C80 DSP is the heart of the SDB. The 'C80 offers the following features:

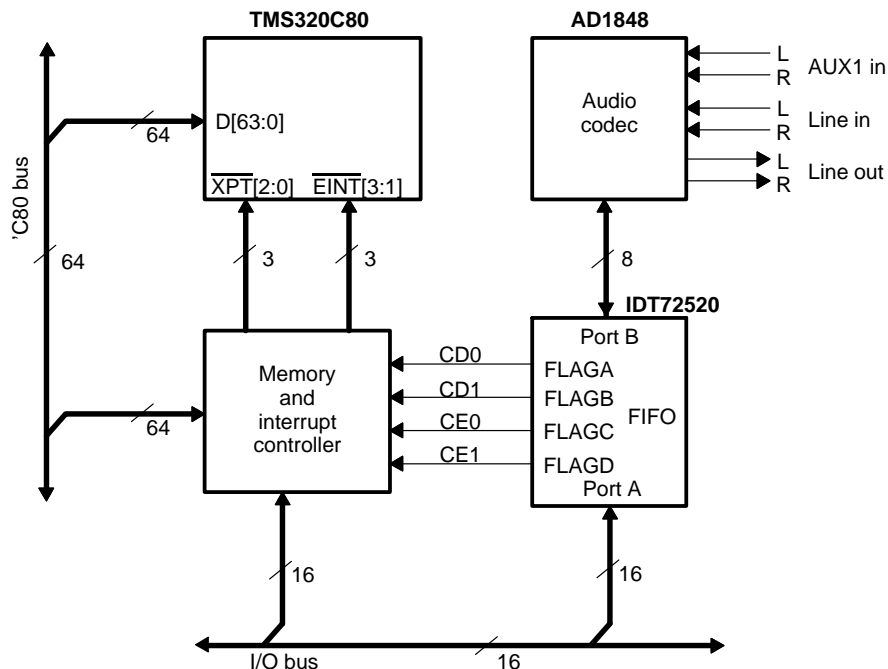
- ☐ Capable of over two billion RISC-like operations per second
- ☐ 40-MHz clock speed
- ☐ 32-bit RISC master processor (MP) with an integrated IEEE-754 floating-point unit and an architecture tuned for efficient compilation of C programs
- ☐ Four 32-bit fixed-point, advanced DSP parallel processors (PPs) in a multiple-instruction, multiple-data (MIMD) configuration
- ☐ Byte-addressable machine with big-endian and little-endian byte ordering support (however, the SDB operates in big-endian mode only)
- ☐ 50K bytes of on-chip SRAM (static random-access memory)
- ☐ On-chip crossbar that allows five instruction fetches and ten parallel data accesses during each cycle to support high transfer rates:
  - 1.8G bytes/s transferring instructions
  - 2.4G bytes/s transferring data
- ☐ On-chip video controller (VC) containing dual frame timers for simultaneous image capture and display
- ☐ 64-bit direct memory access (DMA) transfer controller (TC) capable of up to 400M bytes/s on- and off-chip memory transfers
  - Dynamic sizing of bus width for 64, 32, 16, or 8 bits
  - Access to 64-bit VRAM/RAM/SRAM memory
- ☐ 4G-byte memory address space
- ☐ Synchronous DRAM support
- ☐ Four external interrupts, edge- and level-triggered
- ☐ Built-in emulation features accessed via an IEEE 1149.1-compliant test port
- ☐ Full-scan design (plus boundary scan), accessed via an IEEE 1149.1-compliant test port
- ☐ 3.3-V power supply requirement
- ☐ TI EPIC™ 0.5/0.6-μm CMOS technology
- ☐ Approximately 4 million transistors
- ☐ 305-pin ceramic PGA package

## 2.3 Audio

The audio hardware on the SDB provides everything you need to capture and play back audio data (see Figure 2–2). There are two modes of operation: PIO (programmed input/output) and DMA (direct memory access). In PIO mode, the audio codec's PIO register is accessed to read and write sample data on a sample-by-sample basis. PIO mode does not use the audio FIFO. In DMA mode, samples are transferred (using the DMA controller) between the codec and the audio FIFO. This method of transfer allows audio data to be accessed in chunks rather than sample by sample. DMA is allowed only in one direction at a time, which means full-duplex DMA is not possible.

Most commonly, the audio is set up for DMA operation. For DMA playback, the codec reads data from the FIFO; eventually, the audio FIFO becomes almost empty and asserts one of its flags. This flag generates a 'C80 interrupt in which the interrupt service routine (ISR) issues a packet transfer to write audio data to the FIFO. For DMA capture, the codec writes data to the FIFO; eventually, the FIFO becomes almost full and asserts a flag. This flag triggers a 'C80 interrupt. The ISR then issues a packet transfer to read the audio data from the FIFO.

Figure 2–2. Audio Block Diagram





### 2.3.1 Memory-Mapped Audio Registers

The audio hardware has seven registers accessible on the I/O bus. They are accessed using 16-bit reads or writes. You should use direct external accesses (DEAs) to bypass the MP's data cache. Table 2–1 lists the memory-mapped audio registers. Following the table are diagrams of the register formats and descriptions of the registers and their fields.

*Table 2–1. Audio Registers Summary*

| Register Name | Access     | 'C80 Address | Host Address | Size (Bits) | Description                        |
|---------------|------------|--------------|--------------|-------------|------------------------------------|
| AFIFODAT      | Read/write | 0xE0000200   | 0x2400       | 16          | Audio FIFO data register           |
| CDCIDX        | Read/write | 0xE0000208   | 0x2410       | 8           | Codec index address register       |
| CDCDAT        | Read/write | 0xE000020A   | 0x2414       | 8           | Codec index data register          |
| CDCSTAT       | Read/write | 0xE000020C   | 0x2418       | 8           | Codec status register              |
| CDCPIO        | Read/write | 0xE000020E   | 0x241C       | 8           | Codec PIO data register            |
| AFIFOCFG      | Read/write | 0xE0000210   | 0x2420       | 16          | Audio FIFO configuration register  |
| AFIFOCMD      | Read/write | 0xE0000218   | 0x2430       | 16          | Audio FIFO command/status register |

**AFIFODAT register**

'C80 / host addresses: 0xE0000200 0x2400

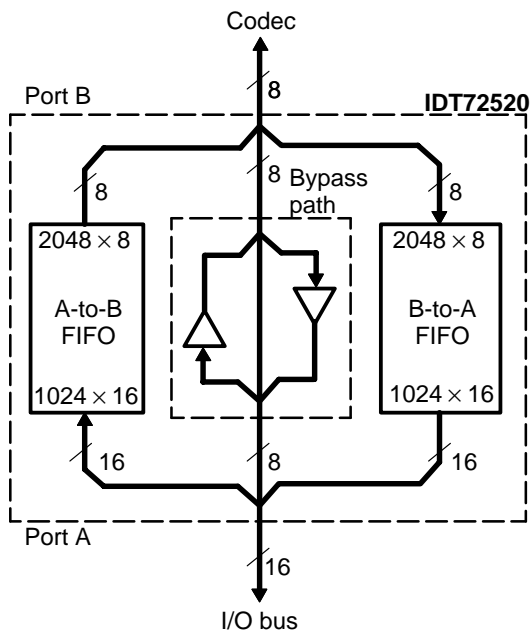
|                          |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |
|--------------------------|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|
| 15                       | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Audio FIFO data register |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |

AFIFODAT is the gateway to the audio FIFO. Reads of this register read a 16-bit word from the B-to-A FIFO; writes to this register write a 16-bit word to the A-to-B FIFO (see Figure 2–3). Normally, packet transfer tables are set up to transfer data to or from this location in blocks. When setting up packet transfer tables, remember that this is a single location and not a range. For example, to transfer 32 16-bit words to the audio FIFO using a packet transfer, you would program the destination parameters of the packet transfer table as follows:

| Parameter Value                  | Meaning                          |
|----------------------------------|----------------------------------|
| Destination Address = 0xE0000200 | This register                    |
| Destination A Count = 2          | 2 bytes, 16-bit register         |
| Destination B Count = 32         | 32 words to transfer             |
| Destination C Count = 0          | 1-dimensional data               |
| Destination B Pitch = 0          | No change to destination address |
| Destination C Pitch = 0          | 1-dimensional data               |

A similar setup should be used when performing packet transfer reads from the audio FIFO.

Figure 2–3. Audio FIFO Block Diagram



**CDCIDX register**

'C80 / host addresses: 0xE0000208 0x2410

|      |     |     |     |   |   |   |   |
|------|-----|-----|-----|---|---|---|---|
| 7    | 6   | 5   | 4   | 3 | 2 | 1 | 0 |
| INIT | MCE | TRD | IXA |   |   |   |   |

CDCIDX is the audio codec's index address register. Whenever an internal register of the codec needs to be accessed, its internal register address must first be written to this register. Then, a read or a write to the codec index data register (CDCDAT) reads or writes to that internal register. This register also has three other bits of information.

|                 |  |
|-----------------|--|
| <b>INIT</b>     | <b>Initialization bit.</b> This read-only bit is set whenever the codec cannot respond to parallel bus cycles and during codec autocalibration.<br><br>INIT = 0      Codec can respond<br>INIT = 1      Codec cannot respond |
| <b>MCE</b>      | <b>Mode change enable.</b> This bit must be set whenever the current functional mode of the codec changes.<br><br>MCE = 0      Mode change disabled<br>MCE = 1      Mode change enabled                                      |
| <b>TRD</b>      | <b>Transfer request disable.</b> This bit is used to stop DMA transfers when the interrupt status (INT) bit is set. TRD should be cleared to 0 because the codec interrupt pin is not used.                                  |
| <b>IXA(4:0)</b> | <b>Index address bits.</b> These bits make up the 5-bit address of the internal register accessed via CDCDAT.  |

**CDCDAT register**

'C80 / host addresses: 0xE000020A 0x2414

|                           |   |   |   |   |   |   |   |
|---------------------------|---|---|---|---|---|---|---|
| 7                         | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Codec index data register |   |   |   |   |   |   |   |

CDCDAT is the audio codec's index data register. Whenever an internal register of the codec needs to be accessed, its internal register address must first be written to the index register (CDCIDX). Then, a read or a write to CDCDAT reads or writes to the desired internal codec register. All codec internal registers are 8-bit registers.

**CDCSTAT register**

'C80 / host addresses: 0xE000020C 0x2418

| 7    | 6    | 5    | 4    | 3    | 2    | 1    | 0   |
|------|------|------|------|------|------|------|-----|
| CU/L | CL/R | CRDY | SOUR | PU/L | PL/R | PRDY | INT |

CDCSTAT is the audio codec's status register.

|             |   |
|-------------|---|
| <b>CU/L</b> | <b>Capture upper/lower.</b> This bit indicates whether the capture PIO port has the upper or lower byte of a sample.<br>CU/L = 0    Lower byte ready<br>CU/L = 1    Upper byte ready or any 8-bit mode                                    |
| <b>CL/R</b> | <b>Capture left/right sample.</b> This bit indicates whether the capture PIO port has the left- or right-channel data.<br>CL/R = 0    Right channel<br>CL/R = 1    Left channel or mono   |
| <b>CRDY</b> | <b>Capture ready.</b> This bit is set to 1 when the codec has a capture sample ready and is valid only when the codec is set up for PIO capture.<br>CRDY = 0    Do not read from PIO port<br>CRDY = 1    Capture ready; PIO port has data |
| <b>SOUR</b> | <b>Sample overrun/underrun.</b> This bit indicates that the most recent sample was not serviced in time.<br>SOUR = 0    No error<br>SOUR = 1    Overrun or underrun occurred  |
| <b>PU/L</b> | <b>Playback upper/lower.</b> This bit indicates whether the playback PIO port is ready for the upper or lower byte of a sample.<br>PU/L = 0    Lower byte needed<br>PU/L = 1    Upper byte needed or any 8-bit mode                       |
| <b>PL/R</b> | <b>Playback left/right sample.</b> This bit indicates whether the playback PIO port is ready for left- or right-channel data.<br>PL/R = 0    Right channel needed<br>PL/R = 1    Left channel or mono                                     |

|             |   |
|-------------|---|
| <b>PRDY</b> | <b>Playback ready.</b> This bit is set to 1 when the codec is ready for a playback sample and is valid only when the codec is set up for PIO playback.<br><br>PRDY = 0    Do not write to PIO port<br>PRDY = 1    Playback ready; PIO port waiting for data |
| <b>INT</b>  | <b>Interrupt status sticky bit.</b> This bit is set when a codec interrupt occurs. Any write to this register clears the INT bit.<br><br>INT = 0       Interrupt pin in active<br>INT = 1       Interrupt pin active  |

**CDCPIO register**

|  |   |   |   |   |   |   |   |
|--|---|---|---|---|---|---|---|
| 'C80 / host addresses: 0xE000020E 0x241C |   |   |   |   |   |   |   |
| 7  | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Codec PIO data register                  |   |   |   |   |   |   |   |

CDCPIO is the programmed input/output (PIO) port of the audio codec. Writes to this register access the codec's PIO playback register, whereas reads from this register access the codec's PIO capture register.

This register should not be read unless the codec is configured for PIO capture mode and should not be written to unless the codec is configured for PIO playback mode. Without codec PIO configuration, a read/write of this register will fail with unpredictable results.

**AFIFOCFG register**

|  |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |
|--|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|
| 'C80 / host addresses: 0xE0000210 0x2420 |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |
| 15                                       | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Audio FIFO configuration register        |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |

AFIFOCFG is the audio FIFO configuration register. Along with the audio FIFO command register (AFIFOCMD), it is accessible by the IDT72520 bidirectional bus-matching FIFO. The IDT72520 has eight internal configuration registers accessed via the AFIFOCFG register. For more details about the audio FIFO, see subsection 2.3.3, *IDT72520 Bidirectional Bus-Matching FIFO*.

**AFIFOCMD register**

'C80 / host addresses: 0xE0000218 0x2430

|    |    |    |    |        |    |   |   |   |   |   |   |         |   |   |   |
|----|----|----|----|--------|----|---|---|---|---|---|---|---------|---|---|---|
| 15 | 14 | 13 | 12 | 11     | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3       | 2 | 1 | 0 |
|    |    |    |    | opcode |    |   |   |   |   |   |   | operand |   |   |   |

AFIFOCMD is the audio FIFO command register. Along with the audio FIFO configuration register (AFIFOCFG), it is accessible by the IDT72520 bidirectional bus-matching FIFO. The IDT72520 is configured by writing commands to this register. The command word has a 4-bit opcode and a 3-bit operand. For more details about the audio FIFO, see subsection 2.3.3, *IDT72520 Bi-directional Bus-Matching FIFO*.

**2.3.2 Audio Codec**

The audio codec used on the SDB is Analog Devices AD1848. Although the codec has a complete set of internal registers, they are accessible only indirectly. This is accomplished using the four registers that are accessible: CDCIDX, CDCDAT, CDCSTAT, and CDCPIO. The codec operates in either stereo or mono mode.

The audio codec supports the following sampling 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 |

The audio codec data formats are:

- ☐ 8-bit unsigned pulse code modulation (PCM)
- ☐ 8-bit  $\mu$ -law companded
- ☐ 8-bit A-law companded
- ☐ 16-bit twos-complement PCM

### 2.3.3 IDT72520 Bidirectional Bus-Matching FIFO

The audio FIFO is the IDT72520 bidirectional bus-matching FIFO. It is  $2048 \times 8$  on the codec side (port B) and  $1024 \times 16$  on the I/O bus side (port A). The FIFO has two accessible registers: AFIFOCMD and AFIFOCFG. This FIFO has a DMA interface to the audio codec, so transfers between the FIFO and the codec require no 'C80 resources. The DMA interface is one-way; that is, the interface either happens for audio capture or audio playback, but not for both at the same time.

The FIFO has four flags: FLAGA, FLAGB, FLAGC, and FLAGD. These flags are independently configurable to assert upon the introduction of any of the FIFO states (empty, almost empty, full, and almost full). The four FIFO flags are tied to event inputs of the SDB's interrupt controller as follows:

- ☐ FLAGA – CD0
- ☐ FLAGB – CD1
- ☐ FLAGC – CE0
- ☐ FLAGD – CE1

The IDT72520 is configured by writing commands to the audio FIFO command register (AFIFOCMD). The command word has a 4-bit opcode and a 3-bit operand (as shown in the AFIFOCMD register diagram on page 2-13). Table 2-2 lists each command the IDT72520 supports.

Table 2–2. Commands Supported by the IDT72520

| Opcode | Operand | Command | Command Description                                 |
|--------|---------|---------|---|
| 0000   | 001     | 0x0001  | Resets B-to-A FIFO (capture)                        |
| 0000   | 010     | 0x0002  | Resets A-to-B FIFO (playback)                       |
| 0000   | 011     | 0x0003  | Resets both FIFOs                                   |
| 0000   | 100     | 0x0004  | Resets internal DMA circuitry                       |
| 0000   | 111     | 0x0007  | Resets all internal pointers                        |
| 0001   | 000     | 0x0100  | Select configuration register 0                     |
| 0001   | 001     | 0x0101  | Select configuration register 1                     |
| 0001   | 010     | 0x0102  | Select configuration register 2                     |
| 0001   | 011     | 0x0103  | Select configuration register 3                     |
| 0001   | 100     | 0x0104  | Select configuration register 4                     |
| 0001   | 101     | 0x0105  | Select configuration register 5                     |
| 0001   | 110     | 0x0106  | Select configuration register 6                     |
| 0001   | 111     | 0x0107  | Select configuration register 7                     |
| 0110   | 000     | 0x0600  | Set DMA direction to capture (through B-to-A FIFO)  |
| 0110   | 001     | 0x0601  | Set DMA direction to playback (through A-to-B FIFO) |
| 0111   | 000     | 0x0700  | Select FIFO status register format 0                |
| 0111   | 001     | 0x0701  | Select FIFO status register format 1                |

Example 2–1 shows sample C code that resets the FIFO.

Example 2–1. Sample C Code to Reset the Audio FIFO

```

/* macro used to access the registers on the I/O bus */
#define AFIFOCMD NOCACHE_USHORT(*(volatile unsigned short *)0xE0000218)

AFIFOCMD = 0x0003; /* reset both FIFOs */
AFIFOCMD = 0x0004; /* reset internal DMA circuitry */
AFIFOCMD = 0x0007; /* reset internal pointers */

```



The IDT72520 has eight internal configuration registers accessed via the audio FIFO configuration register (AFIFOCFG). To access one of the internal configuration registers, you must first issue a command to the device with opcode = 0001 and the operand equal to the configuration register number. The read of AFIFOCFG reads that internal register, whereas a write to AFIFOCFG writes to that configuration register. To read the device's internal configuration register number 3, you write 0x0103 to the command register (AFIFOCMD), and then read the configuration register (AFIFOCFG). Example 2–2 shows the C code to read configuration register 4, mask off the lower four bits, and then write it back. See Table 2–3 for a listing of each IDT72520 internal configuration register.

*Example 2–2. Accessing Internal Configuration Register 4*

```
/* macros used to access the registers on the I/O bus */
#define AFIFOCFG  NOCACHE_USHORT(*(volatile unsigned short *)0xE0000210)
#define AFIFOCMD  NOCACHE_USHORT(*(volatile unsigned short *)0xE0000218)

USHORT Val;

AFIFOCMD = 0x0104; /* select configuration register 4 */
Val = AFIFOCFG; /* read configuration register 4 */
Val = Val & 0xFFFF0; /* mask off lower 4 bits */
AFIFOCFG = Val; /* write back to configuration register 4 */
```

*Table 2–3. IDT72520 Internal Configuration Registers*

| Register No. | Description   |
|--------------|---|
| 0            | A-to-B FIFO almost empty offset (playback) [valid 0x0000 to 0x03FF] |
| 1            | A-to-B FIFO almost full offset (playback) [valid 0x0000 to 0x03FF]  |
| 2            | B-to-A FIFO almost empty offset (capture) [valid 0x0000 to 0x03FF]  |
| 3            | B-to-A FIFO almost full offset (capture) [valid 0x0000 to 0x03FF]   |
| 4            | Flag pin assignments  |
| 5            | Hardware interface register; should be set to 0x0780                |
| 6            | Reserved; do not read or write to this register                     |
| 7            | Reserved; should be initialized to 0x0000                           |

The flag pin assignment register (configuration register 4) specifies which FIFO conditions assert the four FIFO flag pins (FLAGA, FLAGB, FLAGC, and FLAGD). Each FIFO flag pin can be configured to any of the FIFO conditions listed in Table 2–4. Active low means that the flag pin goes low when asserted. The active-high pins should be used for the SDB because a low-to-high transition on the pin triggers the event on the interrupt controller.

*Table 2–4. FIFO Flag Pin Configurations*

| Bits | FIFO              | Condition    | Polarity    |
|------|-------------------|--------------|-------------|
| 0000 | A to B (playback) | Not empty    | Active low  |
| 0001 | A to B (playback) | Almost empty | Active low  |
| 0010 | A to B (playback) | Full         | Active low  |
| 0011 | A to B (playback) | Almost full  | Active low  |
| 0100 | B to A (capture)  | Empty        | Active low  |
| 0101 | B to A (capture)  | Almost empty | Active low  |
| 0110 | B to A (capture)  | Full         | Active low  |
| 0111 | B to A (capture)  | Almost full  | Active low  |
| 1000 | A to B (playback) | Not empty    | Active high |
| 1001 | A to B (playback) | Almost empty | Active high |
| 1010 | A to B (playback) | Full         | Active high |
| 1011 | A to B (playback) | Almost full  | Active high |
| 1100 | B to A (capture)  | Empty        | Active high |
| 1101 | B to A (capture)  | Almost empty | Active high |
| 1110 | B to A (capture)  | Full         | Active high |
| 1111 | B to A (capture)  | Almost full  | Active high |

For example, you could program the flag pin assignments as follows:

FLAGA – (1001) playback FIFO almost empty, active high

FLAGB – (1111) capture FIFO almost full, active high

FLAGC – (1010) playback FIFO full, active high

FLAGD – (1110) capture FIFO empty, active high

The flag pin assignment register value then becomes:

[ddddccccbbbbaaaa] = [1110101011111001] = 0xEAF9

The flag offsets are set to determine when the flags assert. Table 2–5 contains the truth table of the flags.

*Table 2–5. Truth Table for FIFO Flag Assignments*

| Number of Words in FIFO |             | FIFO Flag Condition |              |              |              |
|-------------------------|-------------|---------------------|--------------|--------------|--------------|
| From                    | To          | Empty               | Almost Empty | Almost Full  | Full         |
| 0                       | 0           | Asserted            | Asserted     | Not asserted | Not asserted |
| 1                       | n           | Not asserted        | Asserted     | Not asserted | Not asserted |
| n + 1                   | D – (m + 1) | Not asserted        | Not asserted | Not asserted | Not asserted |
| D – m                   | D – 1       | Not asserted        | Not asserted | Asserted     | Not asserted |
| D                       | D           | Not asserted        | Not asserted | Asserted     | Asserted     |

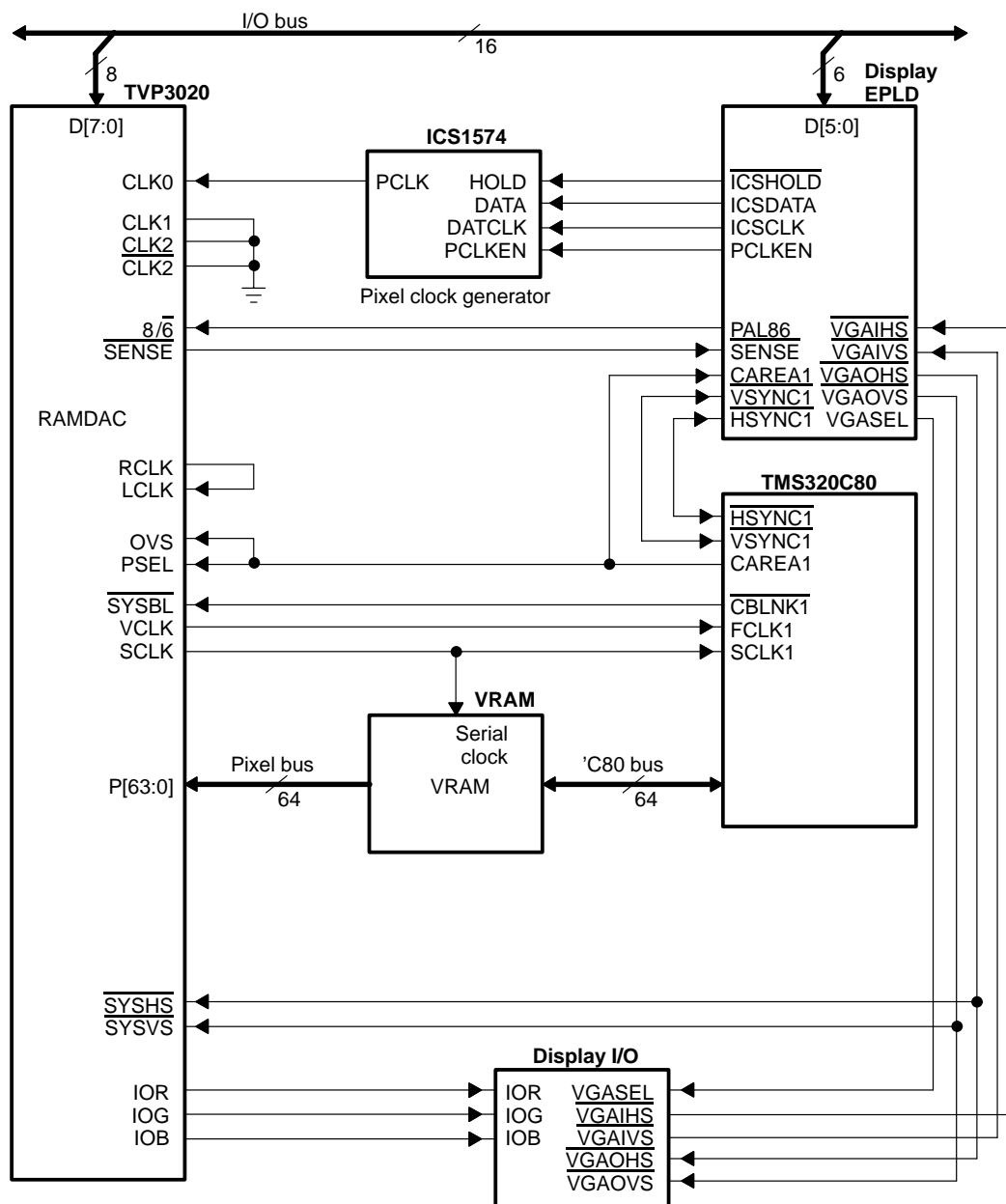
**Legend:** D FIFO depth (1024)  
 m almost-full flag offset  
 n almost-empty flag offset

## 2.4 Video Display

The video display hardware on the SDB provides everything you need to display video and graphics on a standard VGA monitor (see Figure 2–4). It features TI's TVP3020 RAMDAC, ICS1574 programmable pixel clock generator, and analog multiplexing circuitry for video overlay. This is all coupled with 2M bytes of VRAM to support resolutions up to  $1600 \times 1200$  at 8 BPP, noninterlaced. These features' descriptions use the following terms:

|                         |  |
|-------------------------|--|
| <b>Active area</b>      | The area of a display frame that is not in blanking  |
| <b>Pixel</b>            | One picture element (pel)  |
| <b>Pixel resolution</b> | Dimensions of the active area of the display in number of pixels   |
| <b>Dot</b>              | Measurement of time equal to the time required for the display hardware to draw one pixel  |
| <b>Dot rate</b>         | The reciprocal of dot. If it takes 40 ns to display one pixel, then the dot rate is $1/40 \text{ ns} = 25 \text{ MHz}$ . The dot rate is also known as the dot clock frequency ( $F_d$ ) or the pixel clock frequency. |
| <b>Refresh rate</b>     | The number of times a display frame is drawn in one second. A common refresh rate for VGA displays is 60 frames per second. The refresh rate is also known as the vertical frequency ( $F_v$ ).                        |
| <b>Pixel depth</b>      | The number of bits needed to store a pixel in VRAM   |

Figure 2–4. Video Display Block Diagram



### 2.4.1 Memory-Mapped Video Display Registers

The video display hardware has nine registers accessible on the I/O bus. They are accessed using 16-bit reads or writes. You should use direct external accesses (DEAs) to bypass the MP's data cache. Table 2–6 lists the memory-mapped video display registers. Following the table are diagrams of the register formats and descriptions of the registers and their fields.

Table 2–6. Video Display Registers Summary

| Register Name | Access     | 'C80 Address | Host Address | Size (Bits) | Description                    |
|---------------|------------|--------------|--------------|-------------|--------------------------------|
| PALADWR       | Read/write | 0xE0000400   | 0x2800       | 8           | Palette write address register |
| PALHOLD       | Read/write | 0xE0000402   | 0x2804       | 8           | Palette holding register       |
| PELMASK       | Read/write | 0xE0000404   | 0x2808       | 8           | Pixel read-mask register       |
| PALARDR       | Read/write | 0xE0000406   | 0x280C       | 8           | Palette read address register  |
| TVPIDX        | Read/write | 0xE000040C   | 0x2818       | 8           | TVP3020 index register         |
| TVPDAT        | Read/write | 0xE000040E   | 0x281C       | 8           | TVP3020 data register          |
| DIS0CON       | Read/write | 0xE0000410   | 0x2820       | 6           | Display control register 0     |
| DIS1CON       | Read/write | 0xE0000412   | 0x2824       | 6           | Display control register 1     |
| DIS2CON       | Read/write | 0xE0000414   | 0x2828       | 2           | Display control register 2     |

#### ***PALADWR register***

'C80 / host addresses: 0xE0000400 0x2800

|                                |   |   |   |   |   |   |   |
|--------------------------------|---|---|---|---|---|---|---|
| 7                              | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Palette write address register |   |   |   |   |   |   |   |

PALADWR is used to set the write address of the TVP3020 color palette. After this register is set, writes to the palette data holding register (PALHOLD) go into the color palette memory at that address. This register is autoincrementing, so sequential writes to PALHOLD are possible.

**PALHOLD register**

'C80 / host addresses: 0xE0000402 0x2804

|                          |   |   |   |   |   |   |   |
|--------------------------|---|---|---|---|---|---|---|
| 7                        | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Palette holding register |   |   |   |   |   |   |   |

PALHOLD is the TVP3020's palette holding register. Reads of this register perform a read of the color palette RAM, and writes to this register perform a write to the color palette RAM. The color palette RAM must be read or written in triples, that is, in three successive reads or writes. The three reads or writes are a byte of red, a byte of green, and a byte of blue in that order (this is called an *RGB triple*). Upon reading the third byte from this register, the palette read address register (PALADDR) increments by 1. Writing a triple to this register increments the palette write address register (PALADWR) by 1. This allows RGB triples to be read to or written from the palette in sequence, without updating the read or write register each time. Example 2–3 shows the writing of three RGB triples to the color palette RAM starting at RAM address 0x20, and Example 2–4 shows the reading of three RGB triples to the color palette RAM starting at RAM address 0x40.

**Example 2–3. Write Three RGB Triples to the Color Palette RAM**

```
/* macros used to access the registers on the I/O bus */
#define PALADDRD  NOCACHE_USHORT(*(volatile unsigned short *)0xE0000406)
#define PALADWR   NOCACHE_USHORT(*(volatile unsigned short *)0xE0000400)
#define PALHOLD   NOCACHE_USHORT(*(volatile unsigned short *)0xE0000402)

/* set color palette RAM address for writes to 0x20 */
PALADWR = 0x20;

/* remember that all I/O bus accesses must be 16 bit, so the upper 8 bits are
set to zero when writing to 8-bit registers */

/* write first triple (red = 0, green = 0, blue = 0) */
PALHOLD = 0x0000; PALHOLD = 0x0000; PALHOLD = 0x0000;

/* write second triple (red = 0, green = 0, blue = 0) */
PALHOLD = 0x0000; PALHOLD = 0x0000; PALHOLD = 0x0000;

/* write third triple (red = 0, green = 0, blue = 0) */
PALHOLD = 0x0000; PALHOLD = 0x0000; PALHOLD = 0x0000;
```

**Example 2–4. Read Three RGB Triples from the Color Palette RAM**

```

/* macros used to access the registers on the I/O bus */
#define PALADRD  NOCACHE_USHORT(*(volatile unsigned short *)0xE0000406)
#define PALADWR  NOCACHE_USHORT(*(volatile unsigned short *)0xE0000400)
#define PALHOLD  NOCACHE_USHORT(*(volatile unsigned short *)0xE0000402)

/* declare some variables to store RGB triples */
unsigned short R1,G1,B1;
unsigned short R2,G2,B2;
unsigned short R3,G3,B3;

/* set color palette RAM address for reads to 0x40*/
PALADRD = 0x40;

/* remember that I/O bus accesses are 16 bit, so the upper 8 bits need to be
masked off */

/* read first triple */
R1 = PALHOLD & 0x00FF; G1 = PALHOLD & 0x00FF; B1 = PALHOLD & 0x00FF;

/* read second triple */
R2 = PALHOLD & 0x00FF; G2 = PALHOLD & 0x00FF; B2 = PALHOLD & 0x00FF;

/* read third triple */
R3 = PALHOLD & 0x00FF; G3 = PALHOLD & 0x00FF; B3 = PALHOLD & 0x00FF;

```

**PELMASK register**

| 'C80 / host addresses: 0xE0000404 0x2808 |   |   |   |   |   |   |   |
|--|---|---|---|---|---|---|---|
| 7  | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Pixel read-mask register                 |   |   |   |   |   |   |   |

PELMASK is the pixel read-mask register. This 8-bit register is used to enable or disable a bit plane from addressing the color palette RAM in the pseudocolor modes. Each palette address bit is logically ANDed with the corresponding bit from the read-mask register before going to the palette-page register and addressing the palette RAM.



**PALARDR register**

'C80 / host addresses: 0xE0000406 0x280C

|                               |   |   |   |   |   |   |   |
|-------------------------------|---|---|---|---|---|---|---|
| 7                             | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Palette read address register |   |   |   |   |   |   |   |

PALARDR is used to set the read address of the TVP3020 color palette. After this register is set, reads from the palette holding register (PALHOLD) come from the color palette memory at that address. This register is autoincrementing, so sequential reads from PALHOLD are possible.

**TVPIDX register**

'C80 / host addresses: 0xE000040C 0x2818

|                        |   |   |   |   |   |   |   |
|------------------------|---|---|---|---|---|---|---|
| 7                      | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| TVP3020 index register |   |   |   |   |   |   |   |

TVPIDX is the index register to the TVP3020 internal registers. Set this register to access one of the TVP3020 internal registers. For instance, If you want to read the TVP3020 internal register 0x1B, you write 0x1B to TVPIDX and then read from TVPDAT (see Example 2–5).

**TVPDAT register**

'C80 / host addresses: 0xE000040E 0x281C

|                       |   |   |   |   |   |   |   |
|-----------------------|---|---|---|---|---|---|---|
| 7                     | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| TVP3020 data register |   |   |   |   |   |   |   |

TVPDAT is the data port register to the TVP3020 internal registers. Set the internal register address in TVPIDX and then read or write to register TVPDAT, which performs a read or write to the internal register (see Example 2–5).

**Example 2–5. Usage of TVPIDX and TVPDAT Registers**

```
/* macros used to access the registers on the I/O bus */
#define TVPIDX NOCACHE_USHORT(*(volatile unsigned short *)0xE000040C)
#define TVPDAT NOCACHE_USHORT(*(volatile unsigned short *)0xE000040E)

unsigned short Val;

/* set bit zero of TVP3020 register 0x1B */
TVPIDX = 0x1B;
Val = (TVPDAT & 0x00FF) | 0x0001;
TVPDAT = Val;
```

**DIS0CON register**

'C80 / host addresses: 0xE0000410 0x2820

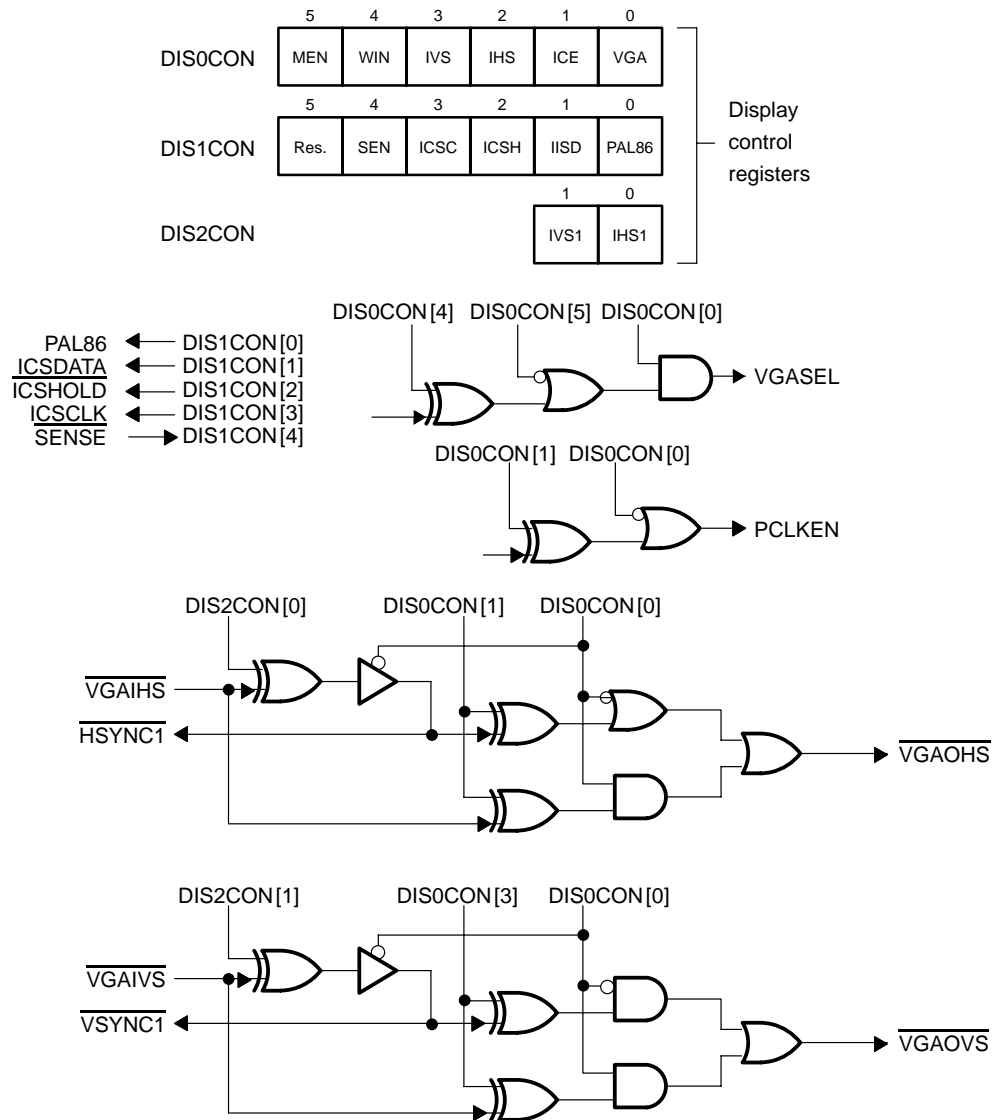
| 5   | 4   | 3   | 2   | 1   | 0   |
|-----|-----|-----|-----|-----|-----|
| MEN | WIN | IVS | IHS | ICE | VGA |

DIS0CON is one of three display control registers on the SDB located in an EPLD (electrically programmable logic device). This register contains bits to control the video overlay feature of the board. A complete description of the video overlay feature is provided in subsection 2.4.4, *Video Overlay Feature (Mixed Mode)*. The display EPLD block diagram depicted in Figure 2–5 shows the logical relationship between the display control registers and the display signals.

|            |  |
|------------|--|
| <b>MEN</b> | <p><b>Mix enable.</b> This bit enables mixed (overlay) mode in which the input from the VGA pass-through cable is mixed in the analog domain with the output of the TVP3020 RAMDAC; that is, VGA is mixed with palette graphics.</p> <p>MEN = 0      Disable mixed mode</p> <p>MEN = 1      Enable mixed mode</p>  |
| <b>WIN</b> | <p><b>Window mode.</b> When the display is in mixed mode (VGA = 1 and MEN = 1), this bit determines which video signal is in the window and which signal is in the background.</p> <p>WIN = 0      Palette graphics window over VGA background</p> <p>WIN = 1      VGA window over palette graphics background</p> |
| <b>IVS</b> | <p><b>Invert vertical sync.</b> Setting this bit to 1 inverts the vertical sync (VS) signal going to the display output connector.</p> <p>IVS = 0      Normal VS</p> <p>IVS = 1      Invert VS</p>   |
| <b>IHS</b> | <p><b>Invert horizontal sync.</b> Setting this bit to 1 inverts the horizontal sync (HS) signal going to the display output connector.</p> <p>IHS = 0      Normal HS</p> <p>IHS = 1      Invert HS</p>   |

|            |  |
|------------|--|
| <b>ICE</b> | <p><b>Invert pixel clock enable.</b> Setting this bit inverts the enable signal (PCLKEN) to the pixel clock generator chip (ICS1574). When the VGA bit is 1, the PCLKEN signal is driven by the horizontal sync signal from the VGA pass-through input (<math>\overline{\text{VGA} \text{HS}}</math>). When the VGA bit is 0, the PCLKEN signal is held high (pixel clock always enabled).</p> <p>ICE = 0      Normal ICS1574 enable</p> <p>ICE = 1      Invert ICS1574 enable</p> |
| <b>VGA</b> | <p><b>VGA/mix enable.</b> When this bit is cleared to 0, the video output of the SDB is palette graphics (that is, the output of the TVP3020 RAMDAC). When this bit is set to 1, either the VGA input from the VGA pass-through cable or a mixture of that and palette graphics is displayed, depending on the MEN (mix enable) bit.</p> <p>VGA = 0      Palette graphics only</p> <p>VGA = 1      VGA or VGA/palette graphics mix enabled</p>                                     |

Figure 2–5. Display EPLD Block Diagram



**DIS1CON register**

'C80 / host addresses: 0xE0000412 0x2824

| 5        | 4   | 3    | 2    | 1    | 0     |
|----------|-----|------|------|------|-------|
| Reserved | SEN | ICSC | ICSH | ICSD | PAL86 |

DIS1CON is one of three display control registers on the SDB located in an EPLD (electrically programmable logic device). This register programs the ICS1574 pixel clock generator chip. The display EPLD block diagram depicted in Figure 2–5 shows the logical relationship between the display control registers and the display signals.

|                 |   |
|-----------------|---|
| <b>Reserved</b> | This bit is not used.   |
| <b>SEN</b>      | <b><math>\overline{\text{SENSE}}</math> from TVP3020.</b> This bit is read only and returns the state of the TVP3020's $\overline{\text{SENSE}}$ line.  |
| <b>ICSC</b>     | <b>ICS1574 clock.</b> This bit is tied directly to the DATCLK pin of the ICS1574 pixel clock generator chip and is used for serial communication to the ICS1574 device. See subsection 2.4.2, <i>ICS1574 Pixel Clock Generator</i> , for more details.                              |
| <b>ICSH</b>     | <b>ICS1574 hold.</b> This bit is tied directly to the $\overline{\text{HOLD}}$ pin of the ICS1574 pixel clock generator chip and is used for serial communication to the ICS1574 device. See subsection 2.4.2, <i>ICS1574 Pixel Clock Generator</i> , for more details.             |
| <b>ICSD</b>     | <b>ICS1574 data.</b> This bit is tied directly to the DATA pin of the ICS1574 pixel clock generator chip and is used for serial communication to the ICS1574 device. See subsection 2.4.2, <i>ICS1574 Pixel Clock Generator</i> , for more details.                                 |
| <b>PAL86</b>    | <p><b>TVP3020 palette 8/6 mode.</b> This bit is only for backward compatibility with earlier graphics systems and is normally not used. It should always be set to 1.</p> <p>PAL86 = 0    6-bit component palette graphics</p> <p>PAL86 = 1    8-bit component palette graphics</p> |

**DIS2CON register**

'C80 / host addresses: 0xE0000414 0x2828

| 1    | 0    |
|------|------|
| IVS1 | IHS1 |

DIS2CON is one of three display control registers on the SDB located in an EPLD (electrically programmable logic device). This register is used to invert the sync signal inputs of the 'C80 when the video display is set for mixed mode. The display EPLD block diagram depicted in Figure 2–5 shows the logical relationship between the display control registers and the display signals.

**IVS1**       **$\overline{\text{VSYNC1}}$  invert.** When the display is set for mixed mode, setting this bit inverts the vertical sync input to the 'C80.

IVS1 = 0      Normal  $\overline{\text{VSYNC1}}$  input to 'C80

IVS1 = 1      Invert  $\overline{\text{VSYNC1}}$  input to 'C80

**IHS1**       **$\overline{\text{HSYNC1}}$  invert.** When the display is set for mixed mode, setting this bit inverts the horizontal sync input to the 'C80.

IHS1 = 0      Normal  $\overline{\text{HSYNC1}}$  input to 'C80

IHS1 = 1      Invert  $\overline{\text{HSYNC1}}$  input to 'C80

## 2.4.2 ICS1574 Pixel Clock Generator

The SDB uses the ICS1574 programmable pixel clock generator chip manufactured by Integrated Systems, Inc. This device is fully programmable to output a clock frequency up to 400 MHz. It contains a crystal oscillator, a phase-frequency detector, a prescaler, and a postscaler. The device has one internal 56-bit register, which is written serially using its DATA,  $\overline{\text{HOLD}}$ , and DATCLK pins. These three pins are tied to the ICSD, ICSH, and ICSC bits, respectively, of the DIS1CON register. Actually a 56-bit shift register accepts the serial input and, when all serial bits are shifted in, the shift register contents are written to the internal register and take effect.

To program the ICS1574, the ICSH bit should go low and remain low until the last bit is written. When ICSH goes high, the data in the shift register is transferred to the internal register. The serial data bit (ICSD) is shifted into the ICS1574 upon the 0-to-1 transition of the DATCLK bit. Four operations are needed to program the device:

- 1) Serially write a 1 bit.
- 2) Serially write a 0 bit.
- 3) Serially write a 1 bit as the last bit.
- 4) Serially write a 0 bit as the last bit.

Example 2–6 shows sample code to do each operation.

### *Example 2–6. Sample Code for Programming the ICS1574*

```
/* macros used to access the registers on the I/O bus */
#define DIS1CON NOCACHE_USHORT(*(volatile unsigned short *)0xE0000412)

/* in each case, the PAL86 bit is set to 1, which has nothing to do with */
/* programming the ICS1574 but is the desired setting */

/* write a 1 bit */
DIS1CON = 0x03; /* ICSC = 0, ICSH = 0, ICSD = 1, PAL86 = 1 */
DIS1CON = 0x0B; /* ICSC = 1, ICSH = 0, ICSD = 1, PAL86 = 1 */

/* write a 0 bit */
DIS1CON = 0x01; /* ICSC = 0, ICSH = 0, ICSD = 0, PAL86 = 1 */
DIS1CON = 0x09; /* ICSC = 1, ICSH = 0, ICSD = 0, PAL86 = 1 */

/* write a 1 bit as the last bit */
DIS1CON = 0x07; /* ICSC = 0, ICSH = 1, ICSD = 1, PAL86 = 1 */
DIS1CON = 0x0F; /* ICSC = 1, ICSH = 1, ICSD = 1, PAL86 = 1 */

/* write a 0 bit as the last bit */
DIS1CON = 0x05; /* ICSC = 0, ICSH = 1, ICSD = 0, PAL86 = 1 */
DIS1CON = 0x0D; /* ICSC = 1, ICSH = 1, ICSD = 0, PAL86 = 1 */
```

The display application programming interface (API) has functionality that translates a frequency into the 56 bits and then writes them out to the device. If you wish to program the ICS1574, you must determine all 56 bits and then write them to the device using the previously described methods.

### 2.4.3 Supported Resolutions

Two factors govern available pixel resolution:

- ☐ Amount of VRAM
- ☐ Refresh rate

The pixel resolution is bound by the amount of VRAM because each pixel has to be stored there. The refresh rate is a limiting factor because the maximum dot rate available on the SDB is 170 MHz, and the dot rate depends on the refresh rate.

The amount of VRAM is determined by pixel resolution and pixel depth. So, for a pixel resolution of  $1600 \times 1200$  with a pixel depth of 8 BPP, 1.92 million bytes of VRAM are needed (see the calculation that follows). The SDB has 2M bytes (2 097 152 bytes) of VRAM, which means there is more than enough storage for a  $1600 \times 1200$  at 8 BPP display.

Calculation for  $1600 \times 1200$  at 8 BPP:

$$\frac{1600 \text{ pixels}}{1 \text{ line}} \times \frac{1200 \text{ lines}}{1 \text{ frame}} \times \frac{8 \text{ bits}}{1 \text{ pixel}} \times \frac{1 \text{ byte}}{8 \text{ bits}} = \frac{1.92 \text{ million bytes}}{1 \text{ frame}}$$

To determine an approximate dot rate, apply two general rules:

- ☐ Approximately 76% of the total horizontal frame width is active, while about 24% is in blanking.
- ☐ Approximately 96% of the total vertical frame height is active, while about 4% is in blanking.

Pixels are displayed only in the active area. Thus, for a  $1600 \times 1200$  pixel resolution, the active area is  $1600 \times 1200$ , whereas the entire frame is about  $2105 \times 1250$  ( $1600/.76 = 2105$ ,  $1200/.96 = 1250$ ). Because a dot is the amount of time it takes to display a pixel, there are  $2105 \times 1250$  dots in the entire frame for this example. Multiply the number of pixels per frame (dots) by the number of frames per second (refresh rate); the result is the number of pixels per second. Pixels per second is the dot rate. So, for a pixel resolution of  $1600 \times 1200$  and a refresh rate of 60 frames per second, a dot rate of 157.9 MHz is needed (as shown in the calculation that follows). The board can handle up to 170 MHz.



Calculation for  $1600 \times 1200$  at 60 frames per second:

$$\frac{1600 \text{ pixels}}{1 \text{ line}} \times \frac{1}{0.76} \times \frac{1200 \text{ lines}}{1 \text{ frame}} \times \frac{1}{0.96} \times \frac{60 \text{ frames}}{1 \text{ s}} = \frac{157.9 \text{ million pixels}}{1 \text{ s}}$$

Apply these calculations to another example, and the result becomes marginal:  $1600 \times 1280$  at 8 BPP and 60 frames per second. There is enough VRAM (2 048 000 bytes), but the dot clock approaches 170 MHz. This dot rate is close to the maximum and could cause slight instability or noise. For this reason, the maximum resolution specified is  $1600 \times 1200$  at 8 BPP with a 60-Hz refresh rate. Table 2–7 specifies the standard resolutions supported by the API.

*Table 2–7. Standard Resolutions Supported by the API*

| Horizontal Resolution (Pixels) | Vertical Resolution (Pixels) | Refresh Rate (Hz) | Pixel Depth (BPP) | Required VRAM (Bytes) | Approximate Required Dot Rate (MHz) |
|--------------------------------|------------------------------|-------------------|-------------------|-----------------------|-------------------------------------|
| 640                            | 480                          | 60                | 8                 | 307 200               | 25.26                               |
| 640                            | 480                          | 60                | 16                | 614 400               | 25.26                               |
| 640                            | 480                          | 60                | 32                | 1 228 800             | 25.26                               |
| 640                            | 480                          | 72                | 8                 | 307 200               | 30.32                               |
| 640                            | 480                          | 72                | 16                | 614 400               | 30.32                               |
| 640                            | 480                          | 72                | 32                | 1 228 800             | 30.32                               |
| 800                            | 600                          | 60                | 8                 | 480 000               | 39.47                               |
| 800                            | 600                          | 60                | 16                | 960 000               | 39.47                               |
| 800                            | 600                          | 60                | 32                | 1 920 000             | 39.47                               |
| 1024                           | 768                          | 60                | 8                 | 786 430               | 64.67                               |
| 1024                           | 768                          | 60                | 16                | 1 572 864             | 64.67                               |
| 1024                           | 768                          | 70                | 8                 | 786 430               | 75.45                               |
| 1024                           | 768                          | 70                | 16                | 1 572 864             | 75.45                               |
| 1280                           | 1024                         | 60                | 8                 | 1 310 720             | 107.79                              |
| 1600                           | 1200                         | 60                | 8                 | 1 920 000             | 157.89                              |

It is clear from Table 2–7 that the display resolution is dependent on the amount of VRAM available. There must be enough VRAM to store the pixels that fill the active area of the display. One feature the API supports is the ability to set up a *display window*. A display window is nothing more than the widening of the blanking area to shrink the active area of the display. Reducing the active area size reduces the number of pixels in the active area, thus reducing the VRAM storage requirements.

When the API sets up a display window, the monitor timing parameters are calculated for a full resolution window. The active area then is shrunk to the window size and position. One might set up a display resolution of  $1024 \times 768$  and then specify a window of  $512 \times 512$ . The dot clock and sync signal parameters are calculated for a  $1024 \times 768$  resolution to allow the monitor to sync up. Then the blanking signals are programmed such that an active area of  $512 \times 512$  is produced. By setting up a display window, you can achieve higher resolutions for a given pixel size. For instance, the SDB does not support  $1024 \times 768$  at 32 BPP, but you could set up a  $640 \times 480$  at 32 BPP window in a  $1024 \times 768$  display. This method is useful when using the video overlay feature.

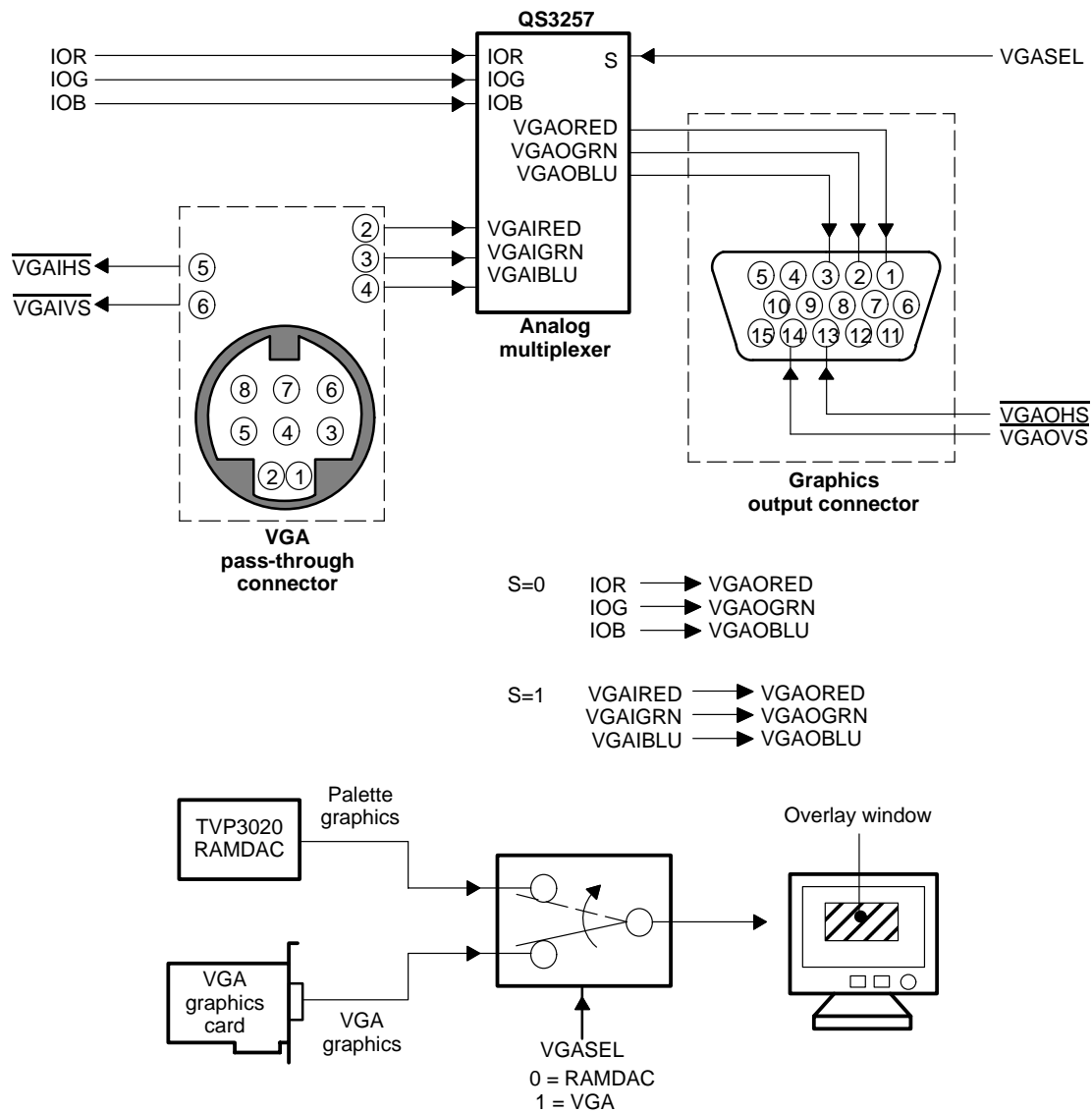
#### 2.4.4 Video Overlay Feature (Mixed Mode)

The SDB supports analog video overlay (mixed mode) by multiplexing between palette graphics (TVP3020 RAMDAC output) and VGA graphics (input from VGA pass-through cable) using an analog multiplexer. This is illustrated in Figure 2–6. One signal, VGASEL, determines which signal (palette or VGA) is sent to the graphics output of the SDB. Overlay is made possible by the fact that VGASEL can be switched in real time on a pixel-by-pixel basis.

You could, for instance, overlay palette graphics on top of VGA graphics. The multiplexer normally switches to pass VGA graphics to the output, but when the pixel location is within the overlay window, the multiplexer switches, passing palette graphics to the output. Then, as the scan line continues, it reaches the end of the window and the multiplexer switches back to VGA graphics. The 'C80's CAREA1 signal is used to control the multiplexer.

It is important for the pixel clock frequency of the RAMDAC to match the pixel clock frequency of the VGA input. The pixel clock frequency of the VGA input is considered an unknown, so trial and error must be used to match it. The general rules described in subsection 2.4.3, *Supported Resolutions*, to calculate the pixel clock frequency usually yield a close match, but some adjustments need to be applied. The display API provides tweak functions for adjusting the video overlay parameters.

Figure 2–6. Display I/O Block Diagram



## 2.5 Video Capture

The video capture front end on the SDB, as depicted in Figure 2–7, resides on a daughter card mounted on the main board. The front end is capable of capturing NTSC or PAL video in CVBS or S-VHS formats. The chipset decodes and then scales the digitized video data. The output of the scaler is configurable to several pixel formats in both YUV and RGB color space. The scaler output is a 32-bit interface to a  $1024 \times 32$  FIFO. The other end of this FIFO ( $512 \times 64$ ) is tied to the 'C80's data bus and supports peripheral data packet transfers to DRAM or VRAM. The scaler performs scaling by pixel dropping and line dropping but has built-in filters to lessen the effects that pixel and line dropping have.

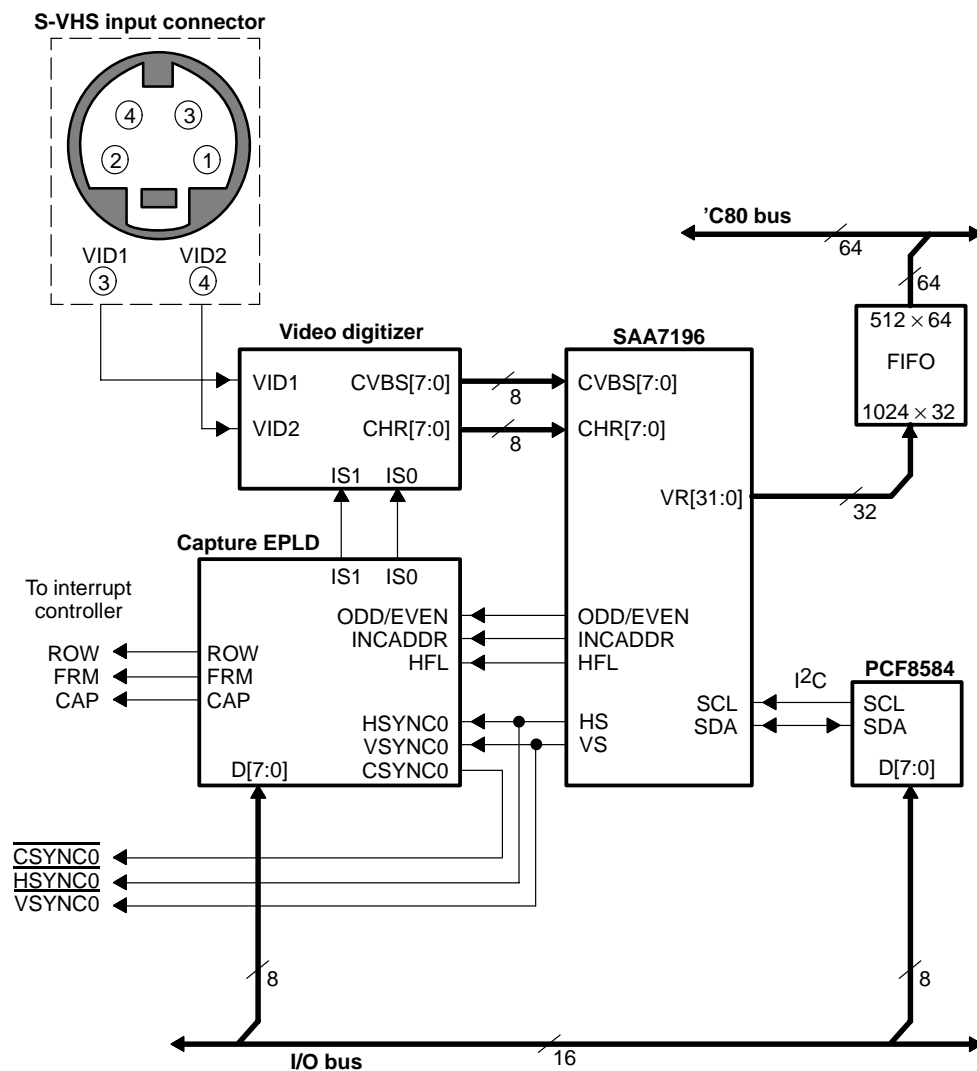
The video capture front end supplies three events for the interrupt controller: FRM, ROW, and CAP. The FRM event occurs at the end of each frame for non-interlaced and at the end of each field for interlaced video. The ROW event occurs at the end of each line of video. CAP is a programmable event that can be triggered by one of eight video capture conditions. Section 2.7, *Interrupt Controller*, provides more detailed information about interrupt controller events and triggers.

### 2.5.1 Video Capture FIFO

The  $512 \times 64$  video capture FIFO, tied directly to the 'C80 data bus (64 bit), allows the 'C80's transfer controller (TC) to packet transfer video data out of the FIFO one whole line at a time. The FIFO is read only but can act as a peripheral to support peripheral data packet transfers to DRAM or VRAM. The video capture FIFO has the following address range:

0xC0400000 to 0xC07FFFFFFF

Figure 2–7. Video Capture Block Diagram



## 2.5.2 Memory-Mapped Video Capture Registers

The video capture hardware has 14 registers accessible on the I/O bus. They must be accessed using 16-bit reads or writes. You should use direct external accesses (DEAs) to bypass the MP's data cache. Table 2–8 lists the memory-mapped video capture registers. Following the table are diagrams of the register formats and descriptions of the registers and their fields.

Table 2–8. Video Capture Registers Summary

| Register Name | Access     | 'C80 Address | Host Address | Size (Bits) | Description                               |
|---------------|------------|--------------|--------------|-------------|---|
| CFIFORST      | Write only | 0xE0000600   | 0x2C00       | 8           | Reset video capture FIFO register         |
| ISRC          | Read only  | 0xE0000602   | 0x2C04       | 2           | Read input source setting register        |
| SVHS          | Write only | 0xE0000608   | 0x2C10       | 8           | Set input to S-VHS register               |
| CVBS1         | Write only | 0xE000060A   | 0x2C14       | 8           | Set input to CVBS1 register               |
| CVBS2         | Write only | 0xE000060C   | 0x2C18       | 8           | Set input to CVBS2 register               |
| CAPID         | Read only  | 0xE000060E   | 0x2C1C       | 8           | Capture card ID register                  |
| INTREG        | Read only  | 0xE0000610   | 0x2C20       | 8           | Interrupt flag register                   |
| INTEN         | Write only | 0xE0000618   | 0x2C24       | 8           | Interrupt enable register                 |
| INTSRC        | Write only | 0xE0000614   | 0x2C28       | 3           | Interrupt source register                 |
| I2CDAT        | Read/write | 0xE0000620   | 0x2C40       | 8           | PCF8584 I <sup>2</sup> C data register    |
| I2CCTRL       | Write only | 0xE0000622   | 0x2C44       | 8           | PCF8584 I <sup>2</sup> C control register |
| I2CSTAT       | Read only  | 0xE0000626   | 0x2C4C       | 8           | PCF8584 I <sup>2</sup> C status register  |
| CAPRST        | Write only | 0xE000063C   | 0x2C78       | 8           | Reset video capture EPLD register         |
| I2CRST        | Write only | 0xE000063C   | 0x2C7C       | 8           | Reset PCF8584 I <sup>2</sup> C controller |

### CFIFORST register

'C80 / host addresses: 0xE0000600 0x2C00

|            |   |   |   |   |   |   |   |
|------------|---|---|---|---|---|---|---|
| 7          | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Pseudobits |   |   |   |   |   |   |   |

CFIFORST is a write-only pseudoregister; that is, a write to this address causes an action to occur, but no data is stored here. Writing any value to this register location resets the video capture FIFO.

**ISRC register**

'C80 / host addresses: 0xE0000602 0x2C04

|              |   |
|--------------|---|
| 1            | 0 |
| Input source |   |

ISRC is a read-only register that returns the current video input configuration. The input configuration is set by writing to the SVHS, CVBS1, or CVBS2 register locations.

ISRC = 00      S-VHS input

ISRC = 01      CVBS1 input

ISRC = 11      CVBS2 input

**SVHS register**

'C80 / host addresses: 0xE0000608 0x2C10

|            |   |   |   |   |   |   |   |
|------------|---|---|---|---|---|---|---|
| 7          | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Pseudobits |   |   |   |   |   |   |   |

SVHS is a write-only pseudoregister; that is, a write to this address causes an action to occur, but no data is stored here. Writing any value to this register location sets up the video capture input for S-VHS. When the setting is S-VHS mode, the luminance is digitized from video input 1, whereas chrominance is digitized from video input 2 (see Figure 2–8).

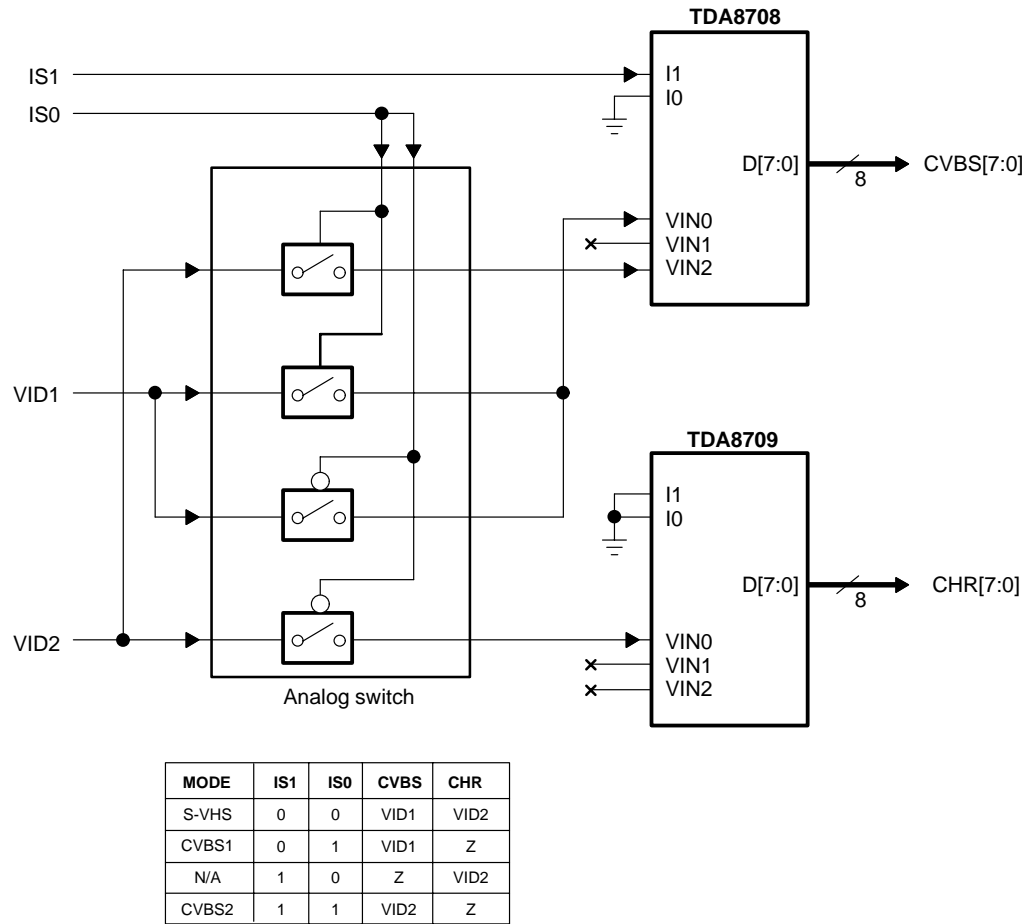
**CVBS1 register**

'C80 / host addresses: 0xE000060A 0x2C14

|            |   |   |   |   |   |   |   |
|------------|---|---|---|---|---|---|---|
| 7          | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Pseudobits |   |   |   |   |   |   |   |

CVBS1 is a write-only pseudoregister; that is, a write to this address causes an action to occur, but no data is stored here. Writing any value to this register location sets up the video capture input for CVBS1. When the setting is CVBS1 mode, the composite video signal at video input 1 is digitized (see Figure 2–8).

Figure 2–8. Video Digitizer Block Diagram



**CVBS2 register**

|  |   |   |   |   |   |   |   |
|--|---|---|---|---|---|---|---|
| 'C80 / host addresses: 0xE000060C 0x2C18 |   |   |   |   |   |   |   |
| 7  | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Pseudobits                               |   |   |   |   |   |   |   |

CVBS2 is a write-only pseudoregister; that is, a write to this address causes an action to occur, but no data is stored here. Writing any value to this register location sets up the video capture input for CVBS2. When the setting is CVBS2 mode, the composite video signal at video input 2 is digitized (see Figure 2–8).



**CAPID register**

'C80 / host addresses: 0xE000060E 0x2C1C

| 7               | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-----------------|---|---|---|---|---|---|---|
| Capture card ID |   |   |   | x | x | x | x |

CAPID is a read-only register that detects the video capture card ID. If reading this register returns 1010 (0xA) in bits 7:4 (the capture card ID), the video capture card is present. Bits 3:0 are ignored.

**INTREG register**

'C80 / host addresses: 0xE0000610 0x2C20

| 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|
| i7 | i6 | i5 | i4 | i3 | i2 | i1 | i0 |

INTREG is a read-only register that returns the video capture interrupt flag bits. Reading this register clears all of its bits. Any of eight video capture conditions can cause an interrupt; each of these conditions has a corresponding bit in INTREG. When one of these conditions occurs, its bit is set in INTREG, assuming the corresponding bit in the interrupt enable register (INTEN) is set.

- i7** Capture FIFO empty flag interrupt flag
- i6** TFRAME interrupt flag
- i5** TROW interrupt flag
- i4** Odd-to-even field transition interrupt flag
- i3** Even-to-odd field transition interrupt flag
- i2**  $\overline{\text{VSYNC0}}$  falling edge interrupt flag
- i1**  $\overline{\text{HSYNC0}}$  falling edge interrupt flag
- i0** Capture FIFO full flag interrupt flag

**INTEN register**

'C80 / host addresses: 0xE0000612 0x2C24

| 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|
| e7 | e6 | e5 | e4 | e3 | e2 | e1 | e0 |

INTEN is a write-only register that enables/disables video capture interrupt sources. Setting a bit to 1 enables the source; clearing it to 0 disables the source. If the enable is set for a source and that source asserts, the corresponding bit in the interrupt flag register (INTREG) is set.

- e7** Capture FIFO empty flag enable flag
- e6** TFRAME enable flag
- e5** TROW enable flag
- e4** Odd-to-even field transition enable flag
- e3** Even-to-odd field transition enable flag
- e2**  $\overline{\text{VSYNC0}}$  falling edge enable flag
- e1**  $\overline{\text{HSYNC0}}$  falling edge enable flag
- e0** Capture FIFO full flag enable flag

**INTSRC register**

'C80 / host addresses: 0xE0000614 0x2C28

| 2          | 1 | 0 |
|------------|---|---|
| CAP source |   |   |

INTSRC is a write-only register that determines which video capture interrupt source triggers the SDB event CAP.

- INTSRC = 000 Capture FIFO full flag interrupt source
- INTSRC = 001  $\overline{\text{HSYNC0}}$  falling edge interrupt source
- INTSRC = 010  $\overline{\text{VSYNC0}}$  falling edge interrupt source
- INTSRC = 011 Even-to-odd field transition interrupt source
- INTSRC = 100 Odd-to-even field transition interrupt source
- INTSRC = 101 TROW interrupt source
- INTSRC = 110 TFRAME interrupt source
- INTSRC = 111 Capture FIFO empty flag interrupt source

Example 2–7 shows how to program CAP to be triggered by an odd-to-even field transition.

**Example 2–7. Programming CAP to Be Triggered by an Odd-to-Even Field Transition**

```
/* macros used to access the registers on the I/O bus */
#define INTREG NOCACHE_USHORT(*(volatile unsigned short *)0xE0000610)
#define INTEN  NOCACHE_USHORT(*(volatile unsigned short *)0xE0000612)
#define INTSRC NOCACHE_USHORT(*(volatile unsigned short *)0xE0000614)

USHORT Junk;

/* read the INTREG register, which clears it */
Junk = INTREG;

/* bind CAP with odd to even field transition */
INTSRC = 0x0004;

/* enable odd to even field transition interrupt condition */
INTEN = 0x0010;
```

**I2CDAT register**

'C80 / host addresses: 0xE0000620 0x2C40

| 7                                      | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|--|---|---|---|---|---|---|---|
| PCF8584 I <sup>2</sup> C data register |   |   |   |   |   |   |   |

I2CDAT, the data register of the PCF8584 I<sup>2</sup>C controller chip, writes data to the I<sup>2</sup>C bus. For more details about the PCF8584 I<sup>2</sup>C controller, see subsection 2.5.4, *PCF8584 I<sup>2</sup>C Bus Controller*.

**I2CCTRL register**

'C80 / host addresses: 0xE0000622 0x2C44

| 7   | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| PCF8584 I <sup>2</sup> C control register |   |   |   |   |   |   |   |

I2CCTRL, the PCF8584 I<sup>2</sup>C control register, controls the operation of the PCF8584 I<sup>2</sup>C controller chip. For more details about the PCF8584 I<sup>2</sup>C controller, see subsection 2.5.4, *PCF8584 I<sup>2</sup>C Bus Controller*.

**I2CSTAT register**

'C80 / host addresses: 0xE0000626 0x2C4C

| 7  | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|--|---|---|---|---|---|---|---|
| PCF8584 I <sup>2</sup> C status register |   |   |   |   |   |   |   |

I2CSTAT is the status register of the PCF8584 I<sup>2</sup>C controller chip. Reading this register returns status information about the PCF8584 I<sup>2</sup>C controller chip. For more details about the PCF8584 I<sup>2</sup>C controller, see subsection 2.5.4, *PCF8584 I<sup>2</sup>C Bus Controller*.

**CAPRST register**

'C80 / host addresses: 0xE000063C 0x2C78

| 7          | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|------------|---|---|---|---|---|---|---|
| Pseudobits |   |   |   |   |   |   |   |

CAPRST is a write-only pseudoregister; that is, a write to this address causes an action to occur, but no data is stored here. Writing any value to this register location resets the video capture EPLD (electrically programmable logic device).

**I2CRST register**

'C80 / host addresses: 0xE000063E 0x2C7C

| 7          | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|------------|---|---|---|---|---|---|---|
| Pseudobits |   |   |   |   |   |   |   |

I2CRST is a write-only pseudoregister; that is, a write to this address causes an action to occur, but no data is stored here. Writing any value to this register location resets the PCF8584 I<sup>2</sup>C controller. For more details about the PCF8584 I<sup>2</sup>C controller, see subsection 2.5.4, *PCF8584 I<sup>2</sup>C Bus Controller*.

### **2.5.3 SAA7196 Video Decoder/Scaler (DESC)**

The video capture front end uses a Philips chipset for digitizing, decoding, and scaling video input. (See Figure 2–7 and Figure 2–8.) Analog video is digitized and then goes into the SAA7196. The SAA7196 is where the digitized video is decoded and then scaled. Next, the scaled output of the SAA7196 is written into the video capture FIFO, where the 'C80 has access to it. All register accesses to the SAA7196 occur via the PCF8584 I<sup>2</sup>C controller.

### **2.5.4 PCF8584 I<sup>2</sup>C Bus Controller**

The PCF8584 I<sup>2</sup>C controller writes to the register space of the SAA7196 video decoder/scaler. Three registers, I2CDAT, I2CCTRL, and I2CSTAT, control the PCF8584 I<sup>2</sup>C device. The API handles all I<sup>2</sup>C bus communications, so the strict protocols used to communicate over the bus are a concern for an application only in rare instances.

## 2.6 Memory Controller

A memory controller on the SDB manages transfers between the 'C80 bus, I/O bus, and the PCI bus. The system DRAM and VRAM are connected to the 'C80 bus. The video capture FIFO also ties to the memory controller.

Integrated with the memory controller is the interrupt controller, which is described in Section 2.7, *Interrupt Controller*.

---

**Note:**

The SDB operates in big-endian mode only.

---

### 2.6.1 DRAM

The SDB has 8M bytes of 64-bit-wide, byte-addressable DRAM (dynamic RAM), which is used for program and data storage. The DRAM is accessed in page mode at two cycles per column for writes and three cycles per column for reads. For a 40-MHz 'C80, this gives a peak transfer rate of 160M bytes/s for writes and 106.66M bytes/s for reads. The DRAM is on two 4M-byte DIMM modules. DIMM rather than SIMM modules were used to save space. The memory controller is hardwired for 8M bytes of DRAM, so it is not upgradable. The DRAM interface supports peripheral data packet transfers (PDPTs) from the video capture FIFO. To use PDPTs to transfer data from the video capture FIFO to DRAM, the 28th bit of the DRAM address must be set. The DRAM has the following address range:

Normal access      0x80000000 to 0x807FFFFF

PDPT access        0x90000000 to 0x907FFFFF

## 2.6.2 VRAM

The SDB has 2M bytes of 64-bit-wide, byte-addressable VRAM (video RAM), which is used for video frame storage. The VRAM is accessed in page mode at two cycles per column for writes and three cycles per column for reads. For a 40-MHz 'C80, this gives a peak transfer rate of 160M bytes/s for writes and 106.66M bytes/s for reads. Because the VRAM is surface mounted on the board, it is not upgradable. The VRAM, by nature, is dual ported, having both a 64-bit processor port and a 64-bit serial port. The serial port supports a peak display rate of 340M bytes/s. The VRAM interface supports PDPTs from the video capture FIFO. To use PDPTs to transfer data from the video capture FIFO to VRAM, the 28th bit of the VRAM address must be set. The VRAM memory is composed of four 4M-bit VRAM devices. Each VRAM device is organized as  $512 \times 512 \times 16$  with 256 words of serial access memory (SAM). The VRAM has the following address range:

Normal access      0xC0000000 to 0xC01FFFFFF

PDPT access        0xD0000000 to 0xD01FFFFFF

## 2.6.3 I/O Bus

All peripheral devices are tied to the I/O bus. The I/O bus is 16 bits wide, is non-byte-addressable, and does not operate in page mode. All I/O bus accesses must be 16 bits. If a register of less than 16 bits is read from the I/O bus, the upper bits of the read value should be ignored. When writing to a register of less than 16 bits on the I/O bus, you should set the upper bits to 0. You should access the I/O bus using DEAs to bypass the MP's data cache. To be compatible with all devices on the bus, each access has six wait states added, which amounts to nine column cycles per access (reads and writes). The audio FIFO is connected to the I/O bus, which means all audio data transfers occur at I/O bus rates. The I/O bus has the following address range:

0xE0000000 to 0xEFFFFFFF

## 2.6.4 Video Capture FIFO

The  $512 \times 64$  video capture FIFO, tied directly to the 'C80 data bus (64 bit), allows the 'C80's transfer controller (TC) to packet transfer video data out of the FIFO one whole line at a time. The FIFO is read only but can act as a peripheral to support peripheral data packet transfers to DRAM or VRAM. The video capture FIFO has the following address range:

0xC0400000 to 0xC07FFFFFF

### 2.6.5 PCI Bus

The memory controller manages all accesses to and from the PCI bus through the PCI FIFO. All PCI FIFO accesses are 32 bits wide and require two cycles. This nets an 80M bytes/s peak transfer rate. For a complete description of the PCI bus interface, see Section 2.8, *PCI Interface*.



## 2.7 Interrupt Controller

The interrupt controller is integrated with the memory controller. It detects external event signals and then takes the appropriate action. The following terminology is used in describing the interrupt controller.

|                            |   |
|----------------------------|---|
| <b>Event</b>               | An occurrence in hardware that requires attention. For example, a FIFO becoming empty, thus requiring more data, is an event. The SDB interrupt controller handles 22 events. Each event is given a mnemonic name; for instance, ROW is an event that occurs when the video capture hardware has captured a row of video and needs it to be read from the video capture FIFO. |
| <b>Event signal</b>        | The means by which the interrupt controller is notified that an event has occurred. For instance, the FLAGA pin of the audio FIFO is tied to one of the interrupt controller inputs. When the interrupt controller detects a rising edge on this input, it is said to be signaled.  |
| <b>Triggering an event</b> | When the interrupt controller detects an event signal, the corresponding event is said to be triggered.   |
| <b>Event source</b>        | Device, condition, or action that signals or triggers an event. For instance, the FLAGA pin of the audio FIFO is an event source because it signals the interrupt controller that an event has occurred.  |
| <b>Event destination</b>   | Once an event has been triggered, action must be taken by some device. This device is the event destination. There are three event destinations on the SDB: 'C80 EINT, 'C80 XPT, and host interrupt. For example, a host interrupt is the event destination of the CE0 event. That is, when the CE0 event is triggered, it causes a host interrupt.                           |

|                         |   |
|-------------------------|---|
| <b>Category-1 event</b> | An event whose event destination is configurable. CD0 is a category-1 event because its event destination can be configured to C80 EINT, C80 XPT, or host interrupt. There are 11 category-1 events (see Table 2–9).  |
| <b>Category-2 event</b> | An event that has one fixed event destination. CE0 is a category-2 event because it only causes host interrupts when it is triggered. All category-2 events have the host interrupt as their event destination. That is, category-2 events can only cause host interrupts. There are 11 category-2 events (see Table 2–10). |

Table 2–9. Category-1 Events

| Event | Description  |
|-------|--|
| CD0   | <b>Audio codec event 0.</b> Triggered by audio FIFO FLAGA.   |
| CD1   | <b>Audio codec event 1.</b> Triggered by audio FIFO FLAGB.   |
| CAP   | <b>General-purpose video capture event.</b> Triggered by CAP from the video capture front end.   |
| ROW   | <b>Video capture row event.</b> Triggered by ROW from the video capture front end. ROW events happen at the end of each captured row of video.     |
| FRM   | <b>Video capture frame event.</b> Triggered by FRM from the video capture front end. FRM events happen at the end of each captured field of video. |
| PCI   | <b>PCI event.</b> General-purpose event, which is usually triggered by the host writing to the PCI bit in STFLAG0.                                 |
| BRD   | <b>Block transfer (blt) read event.</b> Triggered by setting the BLR bit in the PCI status register.   |
| BWR   | <b>Block transfer (blt) write event.</b> Triggered by setting the BLW bit in the PCI status register.  |
| EF2   | <b>FIFO2 empty flag event.</b> Triggered by SDB-to-host FIFO becoming empty.   |
| AE2   | <b>FIFO2 almost empty flag event.</b> Triggered by SDB-to-host FIFO becoming almost empty.   |
| AF1   | <b>FIFO1 almost full flag event.</b> Triggered by host-to-SDB FIFO becoming almost full.   |

Table 2–10. Category-2 Events

| Event | Description  |
|-------|--|
| CE0   | <b>Audio codec error event 0.</b> Triggered by audio FIFO FLAGC.                               |
| CE1   | <b>Audio codec error event 1.</b> Triggered by audio FIFO FLAGD.                               |
| FF1   | <b>FIFO1 full flag event.</b> Triggered by the host-to-SDB FIFO becoming full.                 |
| FF2   | <b>FIFO2 full flag event.</b> Triggered by the SDB-to-host FIFO becoming full.                 |
| EF1   | <b>FIFO1 empty flag event.</b> Triggered by the host-to-SDB FIFO becoming empty.               |
| AE1   | <b>FIFO1 almost empty flag event.</b> Triggered by the host-to-SDB FIFO becoming almost empty. |
| AF2   | <b>FIFO2 almost full event.</b> Triggered by the SDB-to-host FIFO becoming almost full.        |
| PGD   | <b>Programming done event.</b> Triggered when all EPLDs have finished programming.             |
| MCI   | <b>Memory controller event.</b> Reserved.  |
| BMI   | <b>Bus master event.</b> Triggered at the start of a bus master operation.                     |
| XLI   | <b>Test bus controller event.</b> Reserved.  |

### 2.7.1 Memory-Mapped Interrupt Controller Registers

To manipulate SDB events, the interrupt controller has 12 registers accessible on the I/O bus. They are accessed using 16-bit reads or writes. You should use direct external accesses (DEAs) to bypass the MP's data cache. Following the table are diagrams of the register formats and descriptions of the registers and their fields.

*Table 2–11. Interrupt Controller Registers Summary*

| Register Name | Access     | 'C80 Address | Host Address | Size (Bits) | Description               |
|---------------|------------|--------------|--------------|-------------|---------------------------|
| ENABLE0       | Read/write | 0xE0000180   | 0x2300       | 16          | Event enable register 0   |
| ENABLE1       | Read/write | 0xE0000182   | 0x2304       | 16          | Event enable register 1   |
| ENABLE2       | Read/write | 0xE0000184   | 0x2308       | 16          | Event enable register 2   |
| EVSTATE       | Read only  | 0xE0000186   | 0x230C       | 16          | Event state register      |
| CLFLAG0       | Write only | 0xE0000188   | 0x2310       | 16          | Clear flag register 0     |
| RDFLAG0       | Read only  | 0xE0000188   | 0x2310       | 16          | Read flag register 0      |
| STFLAG0       | Write only | 0xE000018A   | 0x2314       | 16          | Set flag register 0       |
| CNFLAG0       | Read only  | 0xE000018A   | 0x2314       | 16          | Condition flag register 0 |
| CLFLAG1       | Write only | 0xE000018C   | 0x2318       | 16          | Clear flag register 1     |
| RDFLAG1       | Read only  | 0xE000018C   | 0x2318       | 16          | Read flag register 1      |
| STFLAG1       | Write only | 0xE000018E   | 0x231C       | 16          | Set flag register 1       |
| CNFLAG1       | Read only  | 0xE000018E   | 0x231C       | 16          | Condition flag register 1 |

**ENABLE0 register**

'C80 / host addresses: 0xE0000180 0x2300

| 15 | 14 | 13 | 12 | 11          | 10 | 9           | 8 | 7           | 6 | 5           | 4 | 3           | 2 | 1           | 0 |
|----|----|----|----|-------------|----|-------------|---|-------------|---|-------------|---|-------------|---|-------------|---|
| 0  | 0  | 0  | 0  | PCI(1:0)    |    | FRM(1:0)    |   | ROW(1:0)    |   | CAP(1:0)    |   | CD1(1:0)    |   | CD0(1:0)    |   |
|    |    |    |    | 00 disabled |    | 00 disabled |   | 00 disabled |   | 00 disabled |   | 00 disabled |   | 00 disabled |   |
|    |    |    |    | 01 XPT5     |    | 01 XPT4     |   | 01 XPT3     |   | 01 N/A      |   | 01 XPT2     |   | 01 XPT1     |   |
|    |    |    |    | 10 EINT3    |    | 10 EINT1    |   | 10 EINT2    |   | 10 EINT3    |   | 10 EINT1    |   | 10 EINT2    |   |
|    |    |    |    | 11 N/A      |    | 11 N/A      |   | 11 N/A      |   | 11 Host     |   | 11 Host     |   | 11 Host     |   |

**ENABLE1 register**

'C80 / host addresses: 0xE0000182 0x2304

| 15 | 14 | 13          | 12 | 11          | 10 | 9           | 8 | 7           | 6 | 5           | 4 | 3 | 2 | 1 | 0 |
|----|----|-------------|----|-------------|----|-------------|---|-------------|---|-------------|---|---|---|---|---|
| 0  | 0  | AF1(1:0)    |    | AE2(1:0)    |    | EF2(1:0)    |   | BWR(1:0)    |   | BRD(1:0)    |   | 0 | 0 | 0 | 0 |
|    |    | 00 disabled |    | 00 disabled |    | 00 disabled |   | 00 disabled |   | 00 disabled |   |   |   |   |   |
|    |    | 01 XPT5     |    | 01 XPT4     |    | 01 XPT3     |   | 01 XPT2     |   | 01 XPT1     |   |   |   |   |   |
|    |    | 10 EINT2    |    | 10 EINT3    |    | 10 EINT1    |   | 10 EINT2    |   | 10 EINT3    |   |   |   |   |   |
|    |    | 11 N/A      |    | 11 N/A      |    | 11 N/A      |   | 11 N/A      |   | 11 N/A      |   |   |   |   |   |

The ENABLE0 and ENABLE1 registers contain the enable bits for all category-1 events (see Table 2–9). Each event has two enable bits that configure the event destination. For example, writing 0x0090 to ENABLE1 sets the BWR event to cause EINT2, and it sets the BRD event to cause XPT1. That is, EINT2 is the event destination of the BWR event, and XPT1 is the event destination of the BRD event. Clearing the enable bits to 00 disables the event. Normally, when you set the enable bits of an event, you leave all other bits alone. Example 2–8 shows how to set the CAP enable bits.

**Example 2–8. Setting the CAP Enable Bits**

```
#define ENABLE0 NOCACHE_USHORT(*(volatile unsigned short*)0xE0000180)
/* enable CAP to EINT3 */
ENABLE0 = (ENABLE0 & 0xFFCF) | 0x0020;
```

**ENABLE2 register**

'C80 / host addresses: 0xE0000184 0x2308

| 15 | 14 | 13 | 12 | 11 | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|----|----|----|----|----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0  | 0  | 0  | 0  | 0  | XLI | BMI | MCI | PGD | AF2 | AE1 | EF1 | FF2 | FF1 | CE1 | CE0 |

The ENABLE2 register contains the enable bits for all category-2 events (see Table 2–10). Setting a bit to 1 enables the event; clearing it to 0 disables the event. All category-2 events cause a host interrupt when the event is enabled and triggered. Normally, when you set the enable bits of an event, you leave all other bits alone. Example 2–9 provides sample code to set/clear the bits.

**Example 2–9. Setting/Clearing Category-2 Events**

```
#define ENABLE2 NOCACHE_USHORT(*(volatile unsigned short*)0xE0000184)
/* enable CE1 */
ENABLE2 = (ENABLE2 & 0xFFFFD) | 0x0002;
/* disable AF2 */
ENABLE2 = ENABLE2 & 0xFFBF;
```

**EVSTATE register**

'C80 / host addresses: 0xE0000186 0x230C

|      |           |    |    |          |    |   |   |   |   |   |   |   |   |   |   |
|------|-----------|----|----|----------|----|---|---|---|---|---|---|---|---|---|---|
| 15   | 14        | 13 | 12 | 11       | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| LINT | EINT(3:1) |    |    | XPT(2:0) |    |   | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

EVSTATE is the event state register. The bits in this register provide a real-time (synchronous) image of the 'C80 event pins. The EVSTATE register is read only.

**LINT**                      **State of the 'C80  $\overline{\text{LINT4}}$  pin.** This pin is an active-low, level-sensitive interrupt pin.

LINT = 0       $\overline{\text{LINT4}}$  is asserted (active)

LINT = 1       $\overline{\text{LINT4}}$  is not asserted (not active)

**EINT(3:1)**                      **State of the 'C80  $\overline{\text{EINT}}[3:1]$  pins.** A 0 in one of the bit positions means the corresponding EINT (external interrupt) pin on the 'C80 is low. 'C80 EINTs are triggered on the rising edge of the  $\overline{\text{EINT}}$  pin.

**XPT(2:0)**                      **State of the 'C80  $\overline{\text{XPT}}[2:0]$  pins.** All zeros means that no XPTs (external packet transfers) are being requested.

XPT = 000      no XPTs are being requested

XPT = 001      XPT1 is being requested

XPT = 010      XPT2 is being requested

XPT = 011      XPT3 is being requested

XPT = 100      XPT4 is being requested

XPT = 101      XPT5 is being requested

XPT = 110      XPT6 is being requested

XPT = 111      XPT7 is being requested

**CLFLAG0 (write) / RDFLAG0 (read) register**

'C80 / host addresses: 0xE0000188 0x2310

| 15 | 14 | 13 | 12  | 11  | 10  | 9   | 8   | 7 | 6 | 5   | 4   | 3   | 2   | 1   | 0   |
|----|----|----|-----|-----|-----|-----|-----|---|---|-----|-----|-----|-----|-----|-----|
| 0  | 0  | 0  | AF1 | AE2 | EF2 | BWR | BRD | 0 | 0 | PCI | FRM | ROW | CAP | CD1 | CD0 |

CLFLAG0 / RDFLAG0 is a dual-purpose register. Writes to this register access CLFLAG0; reads from it access RDFLAG0. CLFLAG0 and RDFLAG0 are located at the same address.

The RDFLAG0 register contains event sticky bits or flags. When a category-1 event is triggered, its corresponding sticky bit in this register is set and remains set until cleared by software. (See Table 2–9 for a list of category-1 events.) The one exception to the bit remaining set is when an event is enabled to cause an XPT. If the enabled event causes an XPT, the corresponding sticky bit is cleared automatically when the 'C80 begins the XPT cycle. The flags in this register get set whether the corresponding event is enabled or not.

For example, assume that the CD0 enable bits in ENABLE0 are set to 0 (disabled), and then the CD0 event is triggered. In this case, the CD0 flag bit in RDFLAG0 gets set, but nothing else occurs. In other words, no EINTs, XPTs, or host interrupts are caused. Also, once a sticky bit is set, no further events happen until the sticky bit is cleared. For instance, assume CD0 is enabled to cause EINT2. CD0 is then triggered, which sets the CD0 sticky bit in RDFLAG0, causing EINT2 to occur on the 'C80. If another CD0 event happens before the sticky bit is cleared, another 'C80 EINT2 is not caused. For this reason, the sticky bit must be cleared in the ISR.

The CLFLAG0 register clears the sticky bits in the RDFLAG0 register. Writing a 1 to a bit has no effect; writing a 0 to a bit clears that bit.



**Example 2–10. Clearing an Event**

```

#define CLFLAG0 NOCACHE_USHORT(*(volatile unsigned short*)0xE0000188)
#define RDFLAG0 NOCACHE_USHORT(*(volatile unsigned short*)0xE0000188)

unsigned short flags;

/* read the event flags */
flags = RDFLAG0;

/* does nothing */
CLFLAG0 = 0xFFFF;

/* clear the ROW flag */
CLFLAG0 = 0xFFFF7;

/* clear all flags in RDFLAG0 */
CLFLAG0 = 0x0000;

```

**STFLAG0 (write) / CNFLAG0 (read) register**

'C80 / host addresses: 0xE000018A 0x2314

|    |    |    |     |     |     |     |     |   |   |     |     |     |     |     |     |
|----|----|----|-----|-----|-----|-----|-----|---|---|-----|-----|-----|-----|-----|-----|
| 15 | 14 | 13 | 12  | 11  | 10  | 9   | 8   | 7 | 6 | 5   | 4   | 3   | 2   | 1   | 0   |
| 0  | 0  | 0  | AF1 | AE2 | EF2 | BWR | BRD | 0 | 0 | PCI | FRM | ROW | CAP | CD1 | CD0 |

STFLAG0 / CNFLAG0 is a dual-purpose register. Writes to this register access STFLAG0; reads from it access CNFLAG0. STFLAG0 and CNFLAG0 are located at the same address.

The STFLAG0 register is used to trigger a category-1 event (see Table 2–9). Writing a 1 to one of the bits in this register triggers the corresponding event just as if that event had actually happened. This capability allows the software to simulate event occurrences.

The CNFLAG0 register returns the state of the event signals coming into the interrupt controller. For instance, the CD0 bit reflects the current state of the audio FIFO FLAGA pin.

**CLFLAG1 (write) / RDFLAG1 (read) register**

'C80 / host addresses: 0xE000018C 0x2318

|    |    |    |    |    |     |     |     |     |     |     |     |     |     |     |     |
|----|----|----|----|----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 15 | 14 | 13 | 12 | 11 | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
| 0  | 0  | 0  | 0  | 0  | XLI | BMI | MCI | PGD | AF2 | AE1 | EF1 | FF2 | FF1 | CE1 | CE0 |

CLFLAG1 / RDFLAG1 operates in the same way as the CLFLAG0 / RDFLAG0 register, except the sticky bits (or flags) in this register correspond to category-2 events (see Table 2–10).

**STFLAG1 (write) / CNFLAG1 (read) register**

'C80 / host addresses: 0xE000018E 0x231C

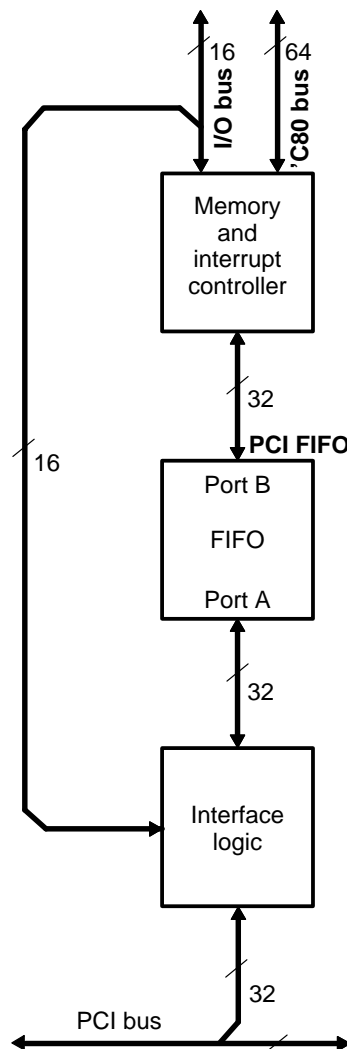
|    |    |    |    |    |     |     |     |     |     |     |     |     |     |     |     |
|----|----|----|----|----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 15 | 14 | 13 | 12 | 11 | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
| 0  | 0  | 0  | 0  | 0  | XLI | BMI | MCI | PGD | AF2 | AE1 | EF1 | FF2 | FF1 | CE1 | CE0 |

STFLAG1 / CNFLAG1 operates in the same way as the STFLAG0 / CNFLAG0 register, except the bits in this register correspond to category-2 events (see Table 2–10).

## 2.8 PCI Interface

The interface between the SDB and the host is a dual-ported FIFO mapped into both 'C80 and host address space. The host side of the FIFO connects directly to the PCI bus through a 32-bit bus transceiver. Bus control logic is contained in EPLDs, which manage all transfers to and from the PCI bus. The SDB side of the FIFO connects to the SDB's memory controller via a 32-bit data path. The memory controller routes FIFO data to or from either the SDB's I/O bus or the 'C80's data bus. Figure 2–9 illustrates this process. The interface logic also generates control signals for the PCI bus itself.

Figure 2–9. PCI Interface Block Diagram



## 2.8.1 PCI Status Register

The host uses the PCI status register (PCISTAT) to monitor the FIFO status, to reset the board, and to cause SDB events. This 32-bit register is accessible only from the host and is located at host address 0x0000. PCISTAT is implemented in EPLD logic. For more details about host addresses, see Section 2.8.5, *Host Address Space*. Following is a diagram of the register's format and definitions of the bit fields.

### *PCISTAT(31:16) register*

Host address: 0x0000

| 31       | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23       | 22 | 21   | 20  | 19  | 18  | 17  | 16  |
|----------|----|----|----|----|----|----|----|----------|----|------|-----|-----|-----|-----|-----|
| Reserved |    |    |    |    |    |    |    | GPI(1:0) |    | PRGD | MB2 | MB1 | AF2 | AF1 | EF1 |

### *PCISTAT(15:0) register*

| 15  | 14       | 13 | 12       | 11 | 10   | 9    | 8   | 7   | 6    | 5         | 4 | 3        | 2 | 1    | 0    |
|-----|----------|----|----------|----|------|------|-----|-----|------|-----------|---|----------|---|------|------|
| EF2 | Reserved |    | GPO(1:0) |    | Res. | BDIS | BLW | BLR | IAEN | FOFF(1:0) |   | FSW(1:0) |   | FRST | MRST |

**Reserved (Res.)** These bits should always be cleared to 0 when writing. The bits are undefined when read.

**GPI(1:0)** **General-purpose input.** These two bits are not used. They should always be written to with 0s.

**PRGD** **Programming done bit.** This bit indicates when EPLD devices are finished programming and the board is ready for use.  
 PRGD = 0 Devices are programming  
 PRGD = 1 Programming is finished

**MB2** **FIFO mailbox2 full flag.** This bit is asserted when the SDB-to-host FIFO mailbox contains data. The host should not attempt to read this mailbox when it is empty, and the host should not attempt to write to this mailbox when it is full.  
 MB2 = 0 SDB-to-host FIFO mailbox has data  
 MB2 = 1 SDB-to-host FIFO mailbox is empty

|            |  |
|------------|--|
| <b>MB1</b> | <p><b>FIFO mailbox1 full flag.</b> This bit is asserted when there is data in the host-to-SDB FIFO mailbox. The 'C80 should not attempt to read this mailbox when it is empty, and the 'C80 should not attempt to write to this mailbox when it is full.</p> <p>MB1 = 0      Host-to-SDB FIFO mailbox has data</p> <p>MB1 = 1      Host-to-SDB FIFO mailbox is empty</p> |
| <b>AF2</b> | <p><b>FIFO2 almost full flag.</b> This bit asserts when the SDB-to-host FIFO becomes almost full. The 'C80 should not attempt to write to this FIFO when it is almost full.</p> <p>AF2 = 0      SDB-to-host FIFO is almost full</p> <p>AF2 = 1      SDB-to-host FIFO is not almost full</p>  |
| <b>AF1</b> | <p><b>FIFO1 almost full flag.</b> This bit asserts when the host-to-SDB FIFO becomes almost full. The host should not attempt to write to this FIFO when it is almost full.</p> <p>AF1 = 0      Host-to-SDB FIFO is almost full</p> <p>AF1 = 1      Host-to-SDB FIFO is not almost full</p>  |
| <b>EF1</b> | <p><b>FIFO1 empty flag.</b> This bit asserts when the host-to-SDB FIFO becomes empty. The 'C80 should not attempt to read this FIFO when it is empty.</p> <p>EF1 = 0      Host-to-SDB FIFO is empty</p> <p>EF1 = 1      Host-to-SDB FIFO is not empty</p>  |
| <b>EF2</b> | <p><b>FIFO2 empty flag.</b> This bit asserts when the SDB-to-host FIFO becomes empty. The host should not attempt to read this FIFO when it is empty.</p> <p>EF2 = 0      SDB-to-host FIFO is empty</p> <p>EF2 = 1      SDB-to-host FIFO is not empty</p>  |

**GPO(1:0) General-purpose output.** These two bits are general-purpose output bits that can be written to by the host, where they can then be used by the SDB. Currently, GPO1 is not used, but GPO0 is used to trigger the PCI event on the SDB's interrupt controller. A 0-to-1 transition of the GPO0 bit triggers the PCI event. The 'C80 code handles the event. Generally, this event is used to cause EINT3 on the 'C80 for message passing to the SDB from the host.

GPO0        0 to 1 transition triggers PCI event

GPO1        Not used

**BDIS Burst disable.** Setting this bit disables all burst transfers between the SDB and PCI bus.

BDIS = 0     Burst transfers enabled

BDIS = 1     Burst transfers disabled

**BLW Block transfer (blt) write event trigger bit.** A 0-to-1 transition of the bit triggers the BWR event on the SDB's interrupt controller. Because block transfers between the SDB and host are common, this feature has been added as an efficient way to trigger an event on the SDB. The 'C80 code handles the BWR event.

**BLR Block transfer (blt) read event trigger bit.** A 0-to-1 transition of the bit triggers the BRD event on the SDB's interrupt controller. Because block transfers between the SDB and host are common, this feature has been added as an efficient way to trigger an event on the SDB. The 'C80 code handles the BRD event.

**IAEN PCI interrupt A enable.** Setting this bit to 1 enables host interrupts.

IAEN = 0     Host interrupts disabled

IAEN = 1     Host interrupts enabled

**FOFF(1:0) PCI FIFO offset.** These bits determine the PCI FIFO offset that governs when the FIFO flags assert/deassert. The suggested setting for the offset is 12 32-bit words.

FOFF = 00 4-word offset

FOFF = 01 8-word offset

FOFF = 10 12-word offset

FOFF = 11 16-word offset

**FSW(1:0) FIFO swap setting bits.** The PCI FIFO logic has functionality to swap bytes on transfers through the FIFO. When a FIFO swap setting change occurs, any data currently in the FIFO gets the new swap setting. This feature is useful for transferring little-endian data from the host to a big-endian format for the SDB. The swapping does not affect the FIFO mailboxes.

FSW = 00 No swapping 0x12345678 → 0x12345678

FSW = 01 Byte swapping 0x12345678 → 0x78563412

FSW = 10 Word swapping 0x12345678 → 0x56781234

FSW = 11 Byte-word swapping 0x12345678 → 0x34127856

**FRST PCI FIFO reset.** Writing a 0 to this bit resets the PCI interface FIFO.

FRST = 0 FIFO reset asserted

FRST = 1 FIFO reset not asserted

**MRST Master reset.** Writing a 0 to this bit pulls the C80's  $\overline{\text{RESET}}$  pin low. Other devices on the board are also tied to this reset signal.

MRST = 0 Master reset asserted

MRST = 1 Master reset not asserted

## 2.8.2 PCI FIFO

The PCI FIFO is depicted in Figure 2–10. The FIFO device has two data FIFOs and two mailboxes. The mailboxes act as single-word FIFOs, meaning that the mailboxes are full after one write and then are empty after one read. The mailboxes can be accessed without disrupting the data in the data FIFOs. Table 2–12 lists the parts of the FIFO device accessed by different host/SDB transfers. All FIFO accesses must be 32-bit accesses.

Figure 2–10. PCI FIFO Block Diagram

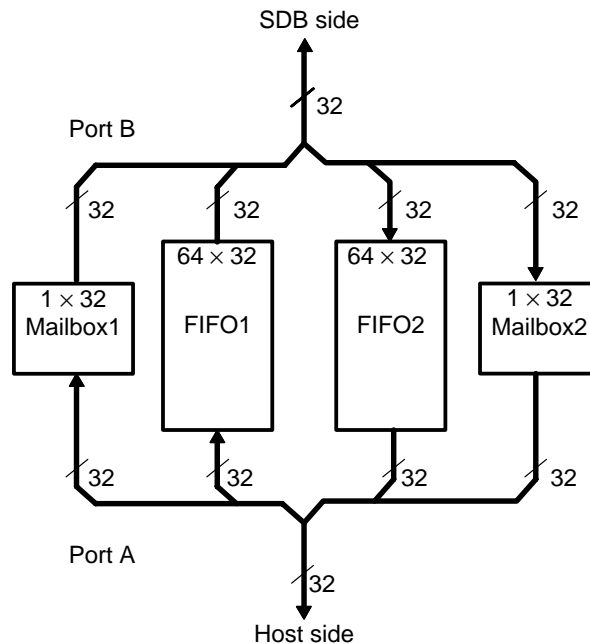


Table 2–12. Parts of the FIFO Device Accessed by Host/SDB Transfers

| Type of Access            | Part of FIFO Device Accessed |                     |
|---------------------------|------------------------------|---------------------|
| 'C80 reads from data FIFO | FIFO1                        | Host-to-SDB FIFO    |
| 'C80 writes to data FIFO  | FIFO2                        | SDB-to-host FIFO    |
| 'C80 reads from mailbox   | Mailbox1                     | Host-to-SDB mailbox |
| 'C80 writes to mailbox    | Mailbox2                     | SDB-to-host mailbox |
| Host reads from data FIFO | FIFO2                        | SDB-to-host FIFO    |
| Host writes to data FIFO  | FIFO1                        | Host-to-SDB FIFO    |
| Host reads from mailbox   | Mailbox2                     | SDB-to-host mailbox |
| Host writes to mailbox    | Mailbox1                     | Host-to-SDB mailbox |



### 2.8.3 TMS320C80 Access to the PCI FIFO

The 'C80 can access the PCI data FIFOs and the PCI FIFO mailboxes. All accesses must be 32-bit accesses. When the 'C80 performs a read from the FIFO data address range, a read is done from the host-to-SDB data FIFO (FIFO1). When the 'C80 performs a write to this range, a write is done to the SDB-to-host data FIFO (FIFO2). The 'C80 should not attempt a read from an empty FIFO or attempt a write to an almost full or full FIFO. When the 'C80 performs a read from the FIFO mailbox address range, a read is done from the host-to-SDB mailbox (mailbox1). When the 'C80 performs a write to this range, a write is done to the SDB-to-host mailbox (mailbox2). The 'C80 should not attempt to read an empty mailbox or attempt to write to a full mailbox.

'C80 FIFO mailbox address range:

0xF8000000 to 0xFBFFFFFFF

'C80 FIFO data address range:

0xFC000000 to 0xFFFFFFFF

### 2.8.4 PCI Plug and Play

The SDB is a PCI plug-and-play device, meaning the host PCI BIOS dynamically configures the board at system boot time. The purpose of plug and play is to dynamically allocate resources to devices so that no two devices use the same resource. These resources include memory address space, I/O address space, interrupts, and DMA channels. Each plug-and-play device requests needed resources during autoconfiguration. The BIOS assigns those resources to the device. Each device has a standard set of PCI configuration registers that the BIOS sets to reflect assigned resources. For the most part, this is all transparent to the user.

The SDB requests 64K bytes of memory address space and one interrupt. During autoconfiguration at boot time, the PCI BIOS detects the SDB request. The PCI BIOS then finds an unused 64K byte chunk of address space and assigns it to the SDB. Next, the PCI assigns an unused interrupt resource to the SDB.

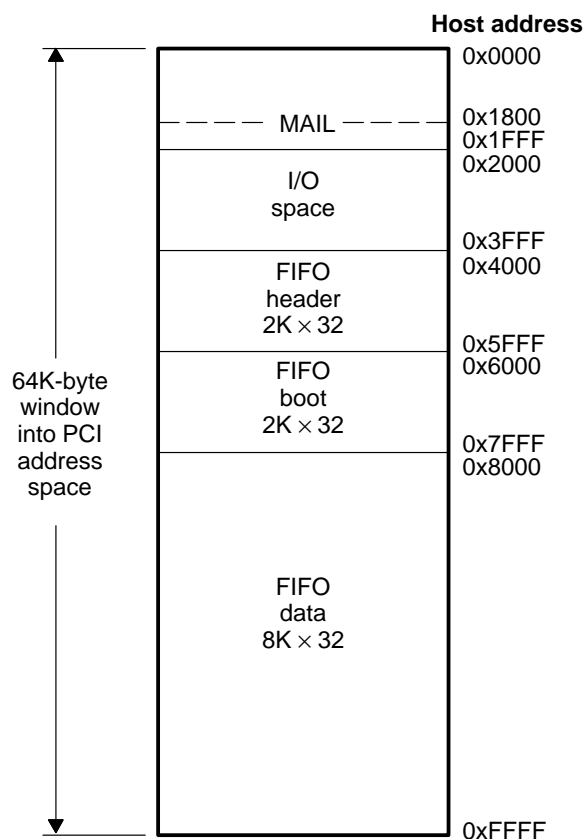
The memory address assigned to the SDB is its physical location on the PCI bus. Generally, an application cannot access this physical address space directly, so a device driver is needed. Because the SDB has 64K bytes of address space, only 16 bits of address are needed to locate any position in that 64K, given the starting address. The device driver locates the starting or physical address by taking a 16-bit address and adding the physical base address. The device driver knows where the physical base address is from the PCI configuration space on the board.

This method of allocating resources allows application code running on the host to locate the board using only a 16-bit address, which the device driver translates to the correct physical address (no matter where PCI assigned the board in its address space). Thus, the 16-bit address remains static since it is just an offset into the 64K address window of PCI address space. This 16-bit offset is referred to as the host address in this guide. For example, assume the PCI BIOS located the SDB at the PCI address of 0xFFBF0000. The device driver translates a 16-bit host address of 0x1234 to a physical location of 0xFFBF1234 on the PCI bus.

## 2.8.5 Host Address Space

The host address space of the SDB is a 64K byte window into the PCI address space. All host addresses to the SDB are specified as 16-bit offsets into this window; the host device driver takes care of the translation (see subsection 2.8.4, *PCI Plug and Play*). The 64K address space of the SDB is partitioned into several areas as depicted in Figure 2–11. Following the diagram is a description of each partition.

Figure 2–11. Host Address Space



|                        |   |
|------------------------|---|
| <b>0x0000</b>          | <b>PCISTAT.</b> This location is the PCI status register and is visible only to the host. (See Section 2.8.1, <i>PCI Status Register</i> , for more information.)   |
| <b>0x1800</b>          | <b>MAIL.</b> This 32-bit location is the port into the FIFO mailboxes and is visible only from the host. Reading this location reads the SDB-to-host mailbox (mailbox2), whereas writes to this location write to the host-to-SDB mailbox (mailbox1). The host should never attempt to read this location when the SDB-to-host mailbox is empty, and the host should never attempt to write to this location when the host-to-SDB mailbox is full. You can determine these conditions by checking the MB1 and MB2 bits in the PCISTAT register. |
| <b>0x2000 – 0x3FFF</b> | <b>I/O space.</b> The entire I/O bus on the SDB is accessible in this address range. Special state machines are implemented that use the FIFO mailboxes to complete I/O bus accesses. The host cannot access the SDB's I/O bus unless both FIFO mailboxes are empty. The transfer is actually a 2-step process handled by the host device driver.   |
| <b>0x4000 – 0x5FFF</b> | <b>FIFO header (2K × 32).</b> This range of addresses maps to the PCI data FIFOs. This range is write-only. A host write to anywhere in this range performs a write to the host-to-SDB data FIFO. The host should not attempt to write to an almost full or full FIFO. The intention of this range is to have a separate FIFO header space to write command server header information. Host reads from this range return the PCI status register (PCISTAT) contents.  |

|                        |  |
|------------------------|--|
| <b>0x6000 – 0x7FFF</b> | <b>FIFO boot (2K × 32).</b> This range of addresses maps to the PCI data FIFOs. This range is write-only. A host write to anywhere in this range performs a write to the host-to-SDB data FIFO. The host should not attempt to write to an almost full or full FIFO. This range provides a separate FIFO boot space to supply boot code to the 'C80. Host reads from this range return the PCI status register (PCISTAT) contents. |
| <b>0x8000 – 0xFFFF</b> | <b>FIFO data (8K × 32).</b> This range of addresses maps to the PCI data FIFOs. A host read from anywhere in this range performs a read from the SDB-to-host data FIFO; a host write to anywhere in this range performs a write to the host-to-SDB data FIFO. The host should not attempt to read an empty FIFO or attempt to write to an almost full or full FIFO. This range provides space for normal FIFO data transfers.      |

### 2.8.6 PCI Bus Mastering

The SDB supports bus master writes to the PCI bus. This means that the SDB has the ability to take control of the PCI bus (master) and write to devices (slaves) on the bus. The destination address of the bus master write is a physical address on the PCI bus. This physical address differs from logical addresses used in PC operating systems such as Windows NT. Translation of a logical address into a physical address and vice versa is beyond the scope of this guide.

Three registers are used to perform bus mastering:

- ☐ Bus master address low (BMAL)
- ☐ Bus master address high (BMAH)
- ☐ Bus master control (BMCTRL)

The application should place the host physical address into BMAL and BMAH (the lower 16 bits into BMAL and the upper 16 bits into BMAH). Once the address registers are configured, setting the BMEN bit of the BMCTRL register starts the bus master transfer. The address registers are autoincrementing during the transfer, but you can disable this feature by setting the increment disable (IDIS) bit of the BMCTRL register.

The bus mastering registers are accessible on the I/O bus. They are accessed using 16-bit reads or writes. You should use direct external accesses (DEAs) to bypass the MP's data cache. Table 2–13 lists the PCI bus mastering registers. Following the table are diagrams of the register formats and descriptions of the registers and their fields.

*Table 2–13. PCI Bus Mastering Registers Summary*

| Register Name | Access     | 'C80 Address | Host Address | Size (Bits) | Description                      |
|---------------|------------|--------------|--------------|-------------|----------------------------------|
| BMAL          | Read/write | 0xE0000A00   | 0x3400       | 16          | Bus master address low register  |
| BMAH          | Read/write | 0xE0000A02   | 0x3404       | 16          | Bus master address high register |
| BMCTRL        | Read/write | 0xE0000104   | 0x2208       | 8           | Bus master control register      |

### ***BMAL register***

'C80 / host addresses: 0xE0000A00 0x3400

|                                 |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |
|---------------------------------|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|
| 15                              | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Bus master address low register |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |

BMAL is the bus master address low register. This register is loaded with the lower 16 bits of the physical PCI address of the target of the bus master operation.

### ***BMAH register***

'C80 / host addresses: 0xE0000A02 0x3404

|                                  |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |
|----------------------------------|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|
| 15                               | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Bus master address high register |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |

BMAH is the bus master address high register. This register is loaded with the upper 16 bits of the physical PCI address of the target of the bus master operation.

**BMCTRL register**

'C80 / host addresses: 0xE0000104 0x2208

|          |   |   |   |      |      |    |   |
|----------|---|---|---|------|------|----|---|
| 7        | 6 | 5 | 4 | 3    | 2    | 1  | 0 |
| Reserved |   |   |   | BMEN | IDIS | b0 |   |

BMCTRL controls the bus mastering feature of the SDB.

**Reserved** Each of these bits should always be cleared to 0.

**BMEN** **Bus mastering enable.** This bit enables bus mastering. When set, the SDB begins transferring data from the SDB-to-host FIFO to the PCI physical address contained in BMAL and BMAH. The transfer operation continues until this bit (BMEN) is cleared. If the FIFO becomes empty, the SDB relinquishes master control of the PCI bus until the 'C80 puts more data into the FIFO, at which point the SDB becomes master again and transfers the data. It is important that you do not clear this bit to complete the bus master operation until all of the data in the FIFO has been transferred.

BMEN = 0 Bus mastering disabled

BMEN = 1 Bus mastering enabled

**IDIS** **Increment disable.** This bit disables the address auto-increment feature of the bus mastering hardware.

IDIS = 0 Address autoincrement enabled

IDIS = 1 Address autoincrement disabled

**b0** **Bit 0.** The bus master hardware requires this bit to be set.

**2.8.7 Bootstrapping**

The term *bootstrapping* refers to the process of resetting the board and providing it some code to run. The SDB is bootstrapped from the host via the PCI FIFO. The 'C80 can be reset by writing a 0 then a 1 to the MRST bit of the PCISTAT register. After it is reset, the 'C80 is halted because the  $\overline{\text{HREQ}}$  pin on the SDB is tied high. To unhalt the 'C80 at this point, the  $\overline{\text{EINT3}}$  signal on the 'C80 must be asserted. The host asserts  $\overline{\text{EINT3}}$  by setting up the interrupt controller and then triggering EINT3 (remember that the host can access the I/O bus without intervention from the 'C80).

When the 'C80 unhalts, it first executes the instruction at 0xFFFFFFFF8. Since the MP's instruction cache is empty, however, the 'C80 must first do an i-cache subblock fill beginning at address 0xFFFFFC0 (a subblock is 64 bytes). This address falls into the address space of the PCI FIFO; therefore, 64 bytes of MP code must be in the FIFO. This means that the host has to put the data in the FIFO from the host side. The MP code then finishes the boot process.



# Audio Capture and Playback API

---

---

---

This chapter discusses the audio macros and data types. It also describes, in alphabetical order, the application programming interface (API) functions associated with the audio capture and playback drivers for the TMS320C8x software development board (SDB).

| <b>Topic</b>  | <b>Page</b> |
|---|-------------|
| <b>3.1 Audio Capture and Playback API Macros and Data Types .....</b> | <b>3-2</b>  |
| <b>3.2 Audio Buffering .....</b>                                      | <b>3-5</b>  |
| <b>3.3 Audio Capture and Playback API Functions .....</b>             | <b>3-9</b>  |

### 3.1 Audio Capture and Playback API Macros and Data Types

Table 3–1 describes the macros used by the audio API and lists the API functions that use each macro. Figure 3–1 provides definitions for the audio API data types. These macros and data types, as well as the API function prototypes, are defined in <audio.h>. The object code resides in sdbdrvs.lib.

Table 3–1. Audio API Macros

*(a) Audio operation mode*

| Macro                  | Value | Description         |
|------------------------|-------|---------------------|
| #define AUDIO_CAPTURE  | 0x02  | DMA capture mode    |
| #define AUDIO_PLAYBACK | 0x03  | DMA playback mode   |
| #define AUDIO_PIO      | 0x04  | Programmed I/O mode |

**Note:** These macros are used by the function Audio\_Install().

*(b) Audio stereo mode*

| Macro                | Value | Description |
|----------------------|-------|-------------|
| #define AUDIO_MONO   | FALSE | Mono mode   |
| #define AUDIO_STEREO | TRUE  | Stereo mode |

**Note:** These macros are used by the function Audio\_Install().

*(c) Audio data coding format*

| Macro               | Value | Description                            |
|---------------------|-------|--|
| #define AUDIO_PCM16 | 0x21  | 16-bit pulse code modulation (signed)  |
| #define AUDIO_PCM8  | 0x12  | 8-bit pulse code modulation (unsigned) |
| #define AUDIO_ALAW8 | 0x13  | 8-bit A-Law                            |
| #define AUDIO_ULAW8 | 0x14  | 8-bit $\mu$ -Law                       |

**Note:** These macros are used by the function Audio\_Install().

*(d) Audio analog input source*

| Macro               | Value | Description                     |
|---------------------|-------|---------------------------------|
| #define AUDIO_LINE  | 0x00  | Line input                      |
| #define AUDIO_AUX1  | 0x01  | Auxiliary 1 input               |
| #define AUDIO_MIXED | 0x03  | Line input with post-DAC mixing |

**Note:** These macros are used by the function Audio\_ProgramInputs().

Table 3–1. Audio API Macros (Continued)

(e) Boolean mute flags for the audio codec

| Macro                | Value | Description    |
|----------------------|-------|----------------|
| #define AUDIO_UNMUTE | FALSE | Unmute channel |
| #define AUDIO_MUTE   | TRUE  | Mute channel   |

**Note:** These macros are used by the following functions:  
 Audio\_ProgramAux1()  
 Audio\_ProgramDacs()

Figure 3–1. Audio API Data Types

(a) Stereo buffer-pointer structure definition

```
typedef struct {
    void *L;      /* pointer to left buffer */
    void *R;      /* pointer to right buffer */
} AUDIO_PTR;
```

(b) Audio metrics structure definition

```
typedef struct {
    BYTE   Mode;      /* audio mode */
    BYTE   Stereo;    /* stereo flag */
    BYTE   Format;     /* data format */
    float  Fs;        /* sample rate */
    BYTE   Bps;       /* bytes per sample */
    BYTE   BlockCt;   /* number of subblocks in a buffer */
    BYTE   BlockSz;   /* number of samples in a subblock */
    BYTE   BuffCt;    /* number of buffers in a queue */
    ULONG  BuffSz;    /* size of buffer in samples */
    ULONG  ByteSz;    /* size of buffer in bytes */
} AUDIO_MET;
```

### **3.1.1 AUDIO\_PTR Data Type**

By encapsulating left and right buffer pointers into one data structure, the AUDIO\_PTR data type simplifies argument passing to audio API functions. The two members of AUDIO\_PTR are L and R, which are both void pointers. Because they are void pointers, the members can point to 8- or 16-bit data buffers. The audio API internally performs the correct type casting of the void pointers to match the buffer data size.

### **3.1.2 Audio Metric Parameters**

The type definition for AUDIO\_MET defines metric parameters of the current audio state. Define a variable of this type, then pass a pointer to it as an argument to Audio\_GetMetrics(). Audio\_GetMetrics() fills in the structure members.

### 3.2 Audio Buffering

This section describes the structure and operation of the internal buffering that is managed by the audio driver. An internal variable of type `BUFFS`, which is local to the audio API module, manages the actual buffers. Figure 3–2 shows the `BUFFS` data type. An application does not have scope into this data type, but it may be important for you to understand the internal workings of the buffering.

Figure 3–2. Internal Audio Buffer Structure

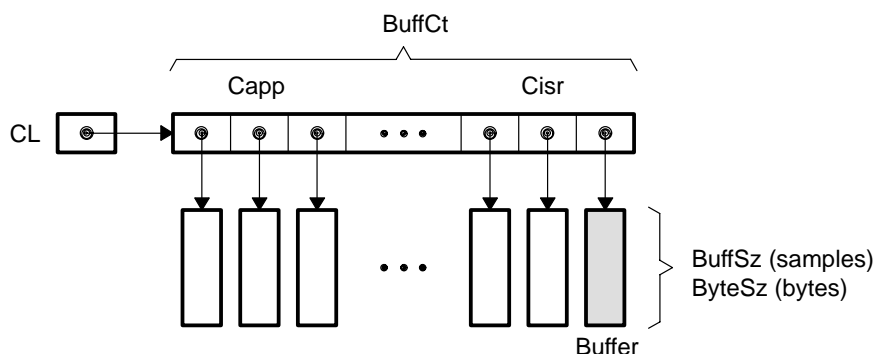
```
typedef struct {
    BYTE    Mode;        /* current audio mode */
    BYTE    Format;       /* current audio format */
    float    Fs;          /* current audio sample rate */
    BYTE    BlockSz;      /* size of each buffer subblock in number of samples */
    BYTE    BlockCt;      /* number of subblocks in each buffer */
    BYTE    BuffCt;       /* number of buffers in each queue */
    ULONG    ByteSz;      /* size of each buffer in bytes */
    ULONG    BuffSz;      /* size of each buffer in samples */
    BYTE    Bps;          /* bytes per sample (1=8-bit, 2=16-bit) */
    BOOL     Stereo;      /* stereo flag (FALSE=mono, TRUE=stereo) */
    BYTE    Capp;         /* applications buffer index into the capture queue */
    BYTE    Cisir;        /* ISRs buffer index into the capture queue */
    BYTE    Papp;         /* applications buffer index into the playback queue */
    BYTE    Pisir;        /* ISRs buffer index into the playback queue */
    BYTE    Cidx;         /* capture subblock index */
    BYTE    Pidx;         /* playback subblock index */
    ULONG    SubSize;     /* size of one subblock in bytes */
    ULONG    Coffset;     /* subblock offset into capture buffer in bytes */
    ULONG    Poffset;     /* subblock offset into playback buffer in bytes */
    void     **CL;        /* pointer to left capture queue */
    void     **CR;        /* pointer to right capture queue */
    void     **PL;        /* pointer to left playback queue */
    void     **PR;        /* pointer to right playback queue */
} BUFFS;
```

The audio buffering structure is created dynamically on the system heap when the application calls `Audio_Install()` and remains intact until the application calls `Audio_UnInstall()`. `Audio_UnInstall()` destroys the queue structure created by `Audio_Install()` and frees up all relative heap storage.

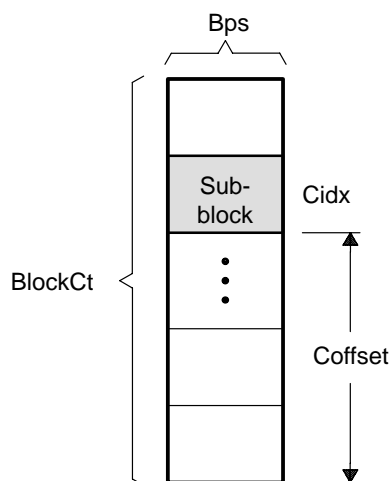
Figure 3–3 illustrates the buffering structure. The main component of the entire buffering mechanism is the queue structure. Individual queues are created for left capture (CL), right capture (CR), left playback (PL), and right playback (PR), depending on the mode (stereo or mono, capture or playback). All pointers within a queue are void pointers that allow 8-bit or 16-bit sample data. The driver takes care of correct type casting of these pointers. Each queue is accessed using one of the void \*\* members of the BUFFS structure. The left capture queue is accessed using CL. Figure 3–3 illustrates the left capture queue only, but the other queues (CR, PL, and PR) have the same structure. The following subsections provide details about the individual queue, individual buffer, and individual subblock depicted in Figure 3–3.

Figure 3–3. Buffering Queue Structure

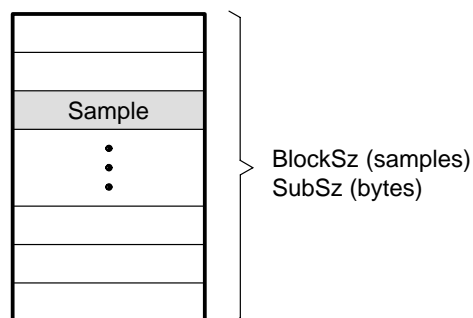
(a) Individual queue



(b) Individual buffer



(c) Individual subblock



### 3.2.1 Individual Queue

Figure 3–3(a) shows an individual queue structure, the left capture queue. The CL member of BUFFS points to an array of void pointers, each of which point to a data buffer. BuffCt indicates the number of such buffers. Each buffer has a size of BuffSz (in number of samples) and ByteSz (in number of bytes). If the audio data is 8 bit (one byte), then the BuffSz is equal to ByteSz.

The Capp and Cisir members of BUFFS are queue indexes into the array of buffer pointers, which range from 0 to *BuffCt* – 1. These indexes continuously advance when audio is running (enabled). When the audio ISR has completed filling a buffer, it advances the Cisir index. If the index overflows ( $\geq \text{BuffCt}$ ), it wraps back to 0. The application advances the Capp index each time `Audio_GetCaptureBufs()` is called. The ISR fills the buffer indexed by Cisir; the buffer returned to the application through a call to `Audio_GetCaptureBufs()` is indexed by Capp. The Capp buffer is locked from the ISR, which gives the application exclusive access to it. When the application calls `Audio_GetCaptureBufs()`, the Capp index is advanced, then a pointer to the indexed buffer is returned.

The Capp and Cisir indexes cannot be equal; that is, the application cannot have access to the same buffer as the ISR. If the application and the ISR were to both have access to the same buffer, the application would read from the buffer as it is being filled by the ISR. This condition introduces the possibility of an index collision. If the ISR tries to advance the Cisir index onto the Capp index, the advance does not take place and the ISR fills the same buffer again. The result is that one buffer of captured audio is skipped. If the application tries to advance the Capp index onto the Cisir index through a call to `Audio_GetCaptureBufs()`, the advance does not take place and the same buffer pointer is returned as the last call. As a result, the application receives the same buffer twice in a row. The only way to prevent index collisions is to have the application advance the Capp index at the same rate (on average) that the ISR advances the Cisir index.

The capture operation described here also applies, in general, to playback operation. The ISR advances Pisir while the application advances Papp through calls to `Audio_GetPlaybackBufs()`. Collisions are handled in a similar fashion.

### 3.2.2 Individual Buffer

Figure 3–3(b) shows the structure of an individual buffer. Bps, or bytes per sample, is the width of the buffer in number of bytes. This value is 1 for 8-bit samples and 2 for 16-bit samples. The application only deals with complete buffers, but the ISR must operate on partitioned buffers. Each partition is called a subblock, and there are BlockCt of them in one buffer.

The ISR manages the index Cidx, which is the subblock index into the current capture buffer. Audio data is packet transferred from the audio FIFO into the audio buffer in chunks equal in size to the subblock. When the audio FIFO becomes almost full (captured audio from the codec), it triggers an interrupt that is handled by the audio capture ISR. This ISR advances the Cidx index, then transfers the new FIFO data into the capture buffer starting at the new subblock offset (Coffset). When the Cidx index overflows, it wraps back to 0 and the next buffer is obtained (Cisr is advanced). The Coffset member is just a byte offset into the buffer used to calculate the destination address of the packet transfer. Each time Cidx is advanced, Coffset is computed. An overview is that the ISR fills up the buffer in chunks and, when it is full, the next one is obtained.

### 3.2.3 Individual Subblock

Figure 3–3(c) depicts the individual subblock. Its size is BlockSz (in number of samples) and SubSz (in number of bytes). Data is transferred from the audio FIFO in chunks equal in size to the subblock size.



### 3.3 Audio Capture and Playback API Functions

Listed below in alphabetical order are the audio capture and playback API functions. Use this list as a table of contents to the audio API functions.

| <b>Function</b>             | <b>Page</b> | <b>Function</b>               | <b>Page</b> |
|-----------------------------|-------------|-------------------------------|-------------|
| Audio_CaptureToMemory ..... | 3-10        | Audio_InstallSema .....       | 3-25        |
| Audio_CodecStat .....       | 3-12        | Audio_PioIn .....             | 3-26        |
| Audio_Disable .....         | 3-13        | Audio_PioOut .....            | 3-27        |
| Audio_Enable .....          | 3-14        | Audio_PioTest .....           | 3-28        |
| Audio_FifoStat .....        | 3-15        | Audio_PlaybackFromMemory ...  | 3-29        |
| Audio_FillBufs .....        | 3-16        | Audio_ProgramAux1 .....       | 3-31        |
| Audio_GetCaptureBufs .....  | 3-17        | Audio_ProgramDacs .....       | 3-32        |
| Audio_GetCodecRegs .....    | 3-18        | Audio_ProgramDigitalMix ..... | 3-33        |
| Audio_GetFifoRegs .....     | 3-19        | Audio_ProgramInputs .....     | 3-34        |
| Audio_GetMetrics .....      | 3-20        | Audio_SetBufferIndexes .....  | 3-35        |
| Audio_GetPlaybackBufs ..... | 3-21        | Audio_SetSampleRate .....     | 3-36        |
| Audio_Init .....            | 3-22        | Audio_UnInstall .....         | 3-37        |
| Audio_Install .....         | 3-23        |                               |             |

|                      |   |
|----------------------|---|
| <b>Function Name</b> | <b>Audio_CaptureToMemory</b>  |
| <b>Syntax</b>        | <code>void Audio_CaptureToMemory(AUDIO_PTR *P, ULONG NumBufs);</code>   |
| <b>Arguments</b>     | <div><div><div>AUDIO_PTR *P</div><div>Pointer to a buffer pointer structure whose elements point to preallocated buffers of memory in which the captured audio is stored</div></div><div><div>ULONG NumBufs</div><div>Amount of audio to capture; value specified as an integral number of DMA buffers as set up by Audio_Install()</div></div></div>   |
| <b>Return Value</b>  | None  |
| <b>Description</b>   | <p>This function captures a specified amount of audio data into buffers pointed to by the members of P. Before calling Audio_CaptureToMemory(), you must first call Audio_Install() to set up the audio hardware for DMA capture. The NumBufs argument specifies how much audio to capture in number of buffers. The buffer size is determined when you call Audio_Install(). If the audio is set up for mono mode, the L member of P is ignored.</p>   |
|                      | <div><b>Notes:</b><ol style="list-style-type: none"><li>1) Do not call Audio_CaptureToMemory() while audio is enabled.</li><li>2) Audio_CaptureToMemory() must be called from a task other than the default task because it waits on a semaphore.</li></ol></div>   |
| <b>Example</b>       | <pre>void AudioTask(void *P) {     AUDIO_PTR Cptr;     AUDIO_MET AM;     BYTE  Format   = AUDIO_PCM16;     BOOL  Stereo   = AUDIO_STEREO;     float Fs       = 48.0;     BYTE  BlockCt  = 20;     BYTE  BlockSz  = 50;     BYTE  BuffCt   = 8;     ULONG NumBufs  = Fs*10;      /* dummy install to set audio metrics */     Audio_Install(AUDIO_CAPTURE, Format, Stereo, Fs, BlockCt,         BlockSz, BuffCt);      /* get the metrics of the new settings */     Audio_GetMetrics(&amp;AM);      /* allocate some DRAM storage for captured audio */     Cptr.L = (void*)memalign(64,AM.ByteSz*NumBufs);     if (Stereo)         Cptr.R = (void*)memalign(64,AM.ByteSz*NumBufs);      /* capture some audio into the allocated buffers */     Audio_CaptureToMemory(&amp;Cptr, NumBufs); }</pre> |

```
/* uninstall the audio settings */
Audio_UnInstall();

/* process the captured audio here */

/* free up the buffer storage */
free((void*)Cptr.L);
    if (Stereo)
        free((void*)Cptr.R);
}
```

|                      |   |       |      |                             |
|----------------------|---|-------|------|-----------------------------|
| <b>Function Name</b> | <b>Audio_CodecStat</b>  |       |      |                             |
| <b>Syntax</b>        | BYTE Audio_CodecStat();   |       |      |                             |
| <b>Arguments</b>     | None  |       |      |                             |
| <b>Return Value</b>  | BYTE  | Bit 7 | CU/L | Capture upper/lower byte    |
|                      |   | Bit 6 | CL/R | Capture left/right channel  |
|                      |   | Bit 5 | CRDY | Capture ready               |
|                      |   | Bit 4 | SOUR | Sample overrun/underrun     |
|                      |   | Bit 3 | PU/L | Playback upper/lower byte   |
|                      |   | Bit 2 | PL/R | Playback left/right channel |
|                      |   | Bit 1 | PRDY | Playback ready              |
|                      |   | Bit 0 | INT  | Interrupt status            |
| <b>Description</b>   | This function reads the audio codec status byte by performing a direct read of the memory-mapped codec status register (CDCSTAT). |       |      |                             |
| <b>Example</b>       | <pre>BYTE stat;<br/><br/>/* read audio codec status */<br/>stat = Audio_CodecStat();</pre>  |       |      |                             |

|                      |   |
|----------------------|---|
| <b>Function Name</b> | <b>Audio_Disable</b>  |
| <b>Syntax</b>        | <code>void Audio_Disable();</code>  |
| <b>Arguments</b>     | None  |
| <b>Return Value</b>  | None  |
| <b>Description</b>   | This function disables the audio subsystem by disabling the audio codec and disabling any associated events.  |
| <b>Example</b>       | <pre>Audio_Install(AUDIO_PLAYBACK, AUDIO_PCM16, AUDIO_STEREO,              48, 16, 64, 4); Audio_Enable();  /* do some audio processing here */  <b>Audio_Disable();</b> Audio_UnInstall();</pre> |

|                      |  |
|----------------------|--|
| <b>Function Name</b> | <b>Audio_Enable</b>  |
| <b>Syntax</b>        | <code>void Audio_Enable();</code>  |
| <b>Arguments</b>     | None   |
| <b>Return Value</b>  | None   |
| <b>Description</b>   | This function enables the audio subsystem by enabling the audio codec and enabling any associated events.  |
| <b>Example</b>       | <pre>Audio_Install(AUDIO_PLAYBACK, AUDIO_PCM16, AUDIO_STEREO, 48,               16, 64, 4); <b>Audio_Enable();</b>  /* do some audio processing here */  Audio_Disable(); Audio_UnInstall();</pre> |

|                      |   |        |                                 |
|----------------------|---|--------|---------------------------------|
| <b>Function Name</b> | <b>Audio_FifoStat</b>   |        |                                 |
| <b>Syntax</b>        | USHORT Audio_FifoStat();  |        |                                 |
| <b>Arguments</b>     | None  |        |                                 |
| <b>Return Value</b>  | USHORT  | Bit 15 | Capture almost empty flag (AEF) |
|                      |   | Bit 14 | Capture FIFO empty flag (FEF)   |
|                      |   | Bit 13 | Playback almost full flag (AFF) |
|                      |   | Bit 12 | Playback FIFO full flag (FFF)   |
|                      |   | Bit 11 | Status register format (1)      |
|                      |   | Bit 10 | Read parity error               |
|                      |   | Bit 9  | Write parity error              |
|                      |   | Bit 8  | Odd byte valid bit              |
|                      |   | Bit 7  | Capture AFF                     |
|                      |   | Bit 6  | Capture FFF                     |
|                      |   | Bit 5  | Playback AEF                    |
|                      |   | Bit 4  | Playback FEF                    |
|                      |   | Bit 3  | DMA direction (0 = capture)     |
|                      |   | Bit 2  | Reserved                        |
|                      |   | Bit 1  | Reserved                        |
|                      |   | Bit 0  | Reserved                        |
| <b>Description</b>   | This function returns the status of the audio FIFO by performing a direct read of the memory-mapped audio FIFO command register (AFIFOCMD). |        |                                 |
|                      | <b>Note:</b><br>Bits 8 and 11 are not used by the software but do occupy bit positions in the status word.                                  |        |                                 |
| <b>Example</b>       | <pre>USHORT stat;<br/><br/>/* read audio FIFO status */<br/>stat = Audio_FifoStat();</pre>  |        |                                 |

|                      |  |
|----------------------|--|
| <b>Function Name</b> | <b>Audio_FillBufs</b>  |
| <b>Syntax</b>        | <code>void Audio_FillBufs(USHORT val);</code>  |
| <b>Arguments</b>     | <code>USHORT val</code> This specifies the fill value. For 8-bit buffers, this value will be cast into a byte.   |
| <b>Return Value</b>  | None   |
| <b>Description</b>   | <p>This function fills the internal audio buffers created by <code>Audio_Install( )</code> with the value specified.</p> <div><b>Note:</b><br/>Do not call <code>Audio_FillBufs()</code> while audio is enabled.</div> |
| <b>Example</b>       | <pre>Audio_Install(AUDIO_PLAYBACK, AUDIO_PCM16, AUDIO_MONO, 8.0,              16, 64, 8); <b>Audio_FillBufs(0x0000);</b> Audio_Enable();  /* do audio processing here */ Audio_Disable();</pre>                        |



|                      |  |   |                                  |
|----------------------|--|---|----------------------------------|
| <b>Function Name</b> | <b>Audio_GetCaptureBufs</b>  |   |                                  |
| <b>Syntax</b>        | BOOL Audio_GetCaptureBufs(AUDIO_PTR *P);   |   |                                  |
| <b>Arguments</b>     | AUDIO_PTR *P   | Pointer to buffer pointer structure that will be filled in by this function with pointers to the captured audio buffers |                                  |
| <b>Return Value</b>  | BOOL   | TRUE  | Internal buffers advanced        |
|                      |  | FALSE   | Internal buffers did not advance |
| <b>Description</b>   | <p>For DMA audio capture, this function is used to get pointers to buffers of captured audio. This function first advances the application index into the capture buffer queue(s), then returns pointers to the newest capture buffers (left and right). If the audio is in mono mode, the R member of P is assigned NULL. If the buffers cannot advance because the ISR is using the next buffers in the queue, this function returns FALSE and pointers to the buffers before advancement are returned. For DMA capture mode, this function must be called, on average, at the same rate the ISR is capturing buffers.</p> |   |                                  |
| <b>Example</b>       | <pre>AUDIO_PTR Cptr;  Audio_Install(AUDIO_CAPTURE, AUDIO_PCM16, AUDIO_STEREO, 8.0,              16, 64, 8); Audio_FillBufs(0x0000); Audio_Enable();  /* loop forever */ while (1) {     TaskWaitSema(AudioSemaId);     <b>Audio_GetCaptureBufs(&amp;Cptr);</b>      /* Cptr.L now points to left captured audio DMA buffer */     /* Cptr.R now points to right captured audio DMA buffer */     /* do audio processing here */ }</pre>  |   |                                  |

**Function Name****Audio\_GetCodecRegs****Syntax**

```
void Audio_GetCodecRegs(BYTE *CodecRegs);
```

**Arguments**

BYTE \*CodecRegs      Pointer to a preallocated array of 16 bytes (unsigned characters) that will be filled with the codec register values

**Return Value**

None

**Description**

This function reads the internal audio codec registers and stores these values into the array pointed to by the CodecRegs argument. The pointer must point to 16 bytes of preallocated memory (16-byte array). The index into the array corresponds to the codec internal register number. None of the internal codec registers are changed. The following list describes the indexes into the codec:

|        |        |                                  |
|--------|--------|----------------------------------|
| Index: | [0x00] | Left input control               |
|        | [0x01] | Right input control              |
|        | [0x02] | Left auxiliary 1 input control   |
|        | [0x03] | Right auxiliary 1 input control  |
|        | [0x04] | Left auxiliary 2 input control   |
|        | [0x05] | Right auxiliary 2 input control  |
|        | [0x06] | Left DAC control                 |
|        | [0x07] | Right DAC control                |
|        | [0x08] | Clock and data format register   |
|        | [0x09] | Interface configuration register |
|        | [0x0A] | Pin control register             |
|        | [0x0B] | Test and initialization register |
|        | [0x0C] | Miscellaneous control register   |
|        | [0x0D] | Digital mix control register     |
|        | [0x0E] | Upper base count register        |
|        | [0x0F] | Lower base count register        |

**Note:**

Use this function for debugging purposes.

**Example 1**

```
BYTE Cregs[16];  
Audio_GetCodecRegs(Cregs);
```

**Example 2**

```
BYTE *Cregs = (BYTE *)malloc(16 * sizeof(BYTE));  
Audio_GetCodecRegs(Cregs);
```

|                      |   |  |        |        |  |  |        |   |  |        |                         |  |        |                         |  |        |                                   |  |        |                                       |
|----------------------|---|--|--------|--------|--|--|--------|---|--|--------|-------------------------|--|--------|-------------------------|--|--------|-----------------------------------|--|--------|---------------------------------------|
| <b>Function Name</b> | <b>Audio_GetFifoRegs</b>  |  |        |        |  |  |        |   |  |        |                         |  |        |                         |  |        |                                   |  |        |                                       |
| <b>Syntax</b>        | void Audio_GetFifoRegs(USHORT *FifoRegs);   |  |        |        |  |  |        |   |  |        |                         |  |        |                         |  |        |                                   |  |        |                                       |
| <b>Arguments</b>     | USHORT *FifoRegs  | Pointer to a preallocated array of 6 USHORTS (unsigned shorts) that will be filled with the FIFO configuration register values |        |        |  |  |        |   |  |        |                         |  |        |                         |  |        |                                   |  |        |                                       |
| <b>Return Value</b>  | None  |  |        |        |  |  |        |   |  |        |                         |  |        |                         |  |        |                                   |  |        |                                       |
| <b>Description</b>   | <p>This function reads the internal audio FIFO configuration registers and stores these values into the array pointed to by the FifoRegs argument. The pointer must point to preallocated memory (6 USHORT array). The index into the array corresponds to the FIFO configuration register number. None of the FIFO configuration registers are changed. The following list describes the indexes into the FIFO:</p> <table><tr><td>Index:</td><td>[0x00]</td><td>Playback FIFO almost empty flag (AEF) offset</td></tr><tr><td></td><td>[0x01]</td><td>Playback FIFO almost full flag (AFF) offset</td></tr><tr><td></td><td>[0x02]</td><td>Capture FIFO AEF offset</td></tr><tr><td></td><td>[0x03]</td><td>Capture FIFO AFF offset</td></tr><tr><td></td><td>[0x04]</td><td>FIFO flag pin assignment register</td></tr><tr><td></td><td>[0x05]</td><td>FIFO interface configuration register</td></tr></table> <div><b>Note:</b><br/>Use this function for debugging purposes.</div> |  | Index: | [0x00] | Playback FIFO almost empty flag (AEF) offset |  | [0x01] | Playback FIFO almost full flag (AFF) offset |  | [0x02] | Capture FIFO AEF offset |  | [0x03] | Capture FIFO AFF offset |  | [0x04] | FIFO flag pin assignment register |  | [0x05] | FIFO interface configuration register |
| Index:               | [0x00]  | Playback FIFO almost empty flag (AEF) offset   |        |        |  |  |        |   |  |        |                         |  |        |                         |  |        |                                   |  |        |                                       |
|                      | [0x01]  | Playback FIFO almost full flag (AFF) offset  |        |        |  |  |        |   |  |        |                         |  |        |                         |  |        |                                   |  |        |                                       |
|                      | [0x02]  | Capture FIFO AEF offset  |        |        |  |  |        |   |  |        |                         |  |        |                         |  |        |                                   |  |        |                                       |
|                      | [0x03]  | Capture FIFO AFF offset  |        |        |  |  |        |   |  |        |                         |  |        |                         |  |        |                                   |  |        |                                       |
|                      | [0x04]  | FIFO flag pin assignment register  |        |        |  |  |        |   |  |        |                         |  |        |                         |  |        |                                   |  |        |                                       |
|                      | [0x05]  | FIFO interface configuration register  |        |        |  |  |        |   |  |        |                         |  |        |                         |  |        |                                   |  |        |                                       |
| <b>Example 1</b>     | <pre>USHORT Fregs[6];<br/>Audio_GetFifoRegs(Fregs);</pre>   |  |        |        |  |  |        |   |  |        |                         |  |        |                         |  |        |                                   |  |        |                                       |
| <b>Example 2</b>     | <pre>USHORT *Fregs = (USHORT *)malloc(6 * sizeof(USHORT));<br/>Audio_GetFifoRegs(Fregs);</pre>  |  |        |        |  |  |        |   |  |        |                         |  |        |                         |  |        |                                   |  |        |                                       |

|                      |   |
|----------------------|---|
| <b>Function Name</b> | <b>Audio_GetMetrics</b>   |
| <b>Syntax</b>        | <code>void Audio_GetMetrics(AUDIO_MET *M);</code>   |
| <b>Arguments</b>     | <p><code>AUDIO_MET *M</code>    Pointer to AUDIO_MET structure that will be filled in by this function</p> <p>AUDIO_MET consists of the following members:</p> <pre>BYTE   Mode;      /* audio mode                */ BYTE   Stereo;    /* stereo flag                */ FBYTE  Formats;   /* data format                */ float  Fs;        /* sample rate                */ BYTE   Bps;       /* bytes per sample           */ BYTE   BlockCt;   /* number of subblocks in a buffer */ BYTE   BlockSz;   /* number of samples in a subblock */ BYTE   BuffCt;    /* number of buffers in a queue  */ ULONG  BuffSz;    /* size of buffer in samples     */ ULONG  ByteSz;    /* size of buffer in bytes       */</pre> |
| <b>Return Value</b>  | None  |
| <b>Description</b>   | <p>This function returns the current operating metrics of the audio subsystem. The AUDIO_MET structure pointed to by M is filled in with the metrics values.</p> <div><b>Note:</b><br/>You can call Audio_GetMetrics() whether audio is enabled or not.</div>   |
| <b>Example</b>       | <pre>AUDIO_MET M;  Audio_GetMetrics(&amp;M);</pre>  |

|                      |  |
|----------------------|--|
| <b>Function Name</b> | <b>Audio_GetPlaybackBufs</b>   |
| <b>Syntax</b>        | BOOL Audio_GetPlaybackBufs(AUDIO_PTR *P);  |
| <b>Arguments</b>     | AUDIO_PTR *P    Pointer to buffer pointer structure whose members (L and R) will be assigned pointers to buffers that need to be filled with audio playback data   |
| <b>Return Value</b>  | BOOL            TRUE            Internal buffers advanced<br>FALSE          Internal buffers did not advance   |
| <b>Description</b>   | <p>For DMA audio playback, this function is used to get buffers for audio playback. This function first advances the application index into the playback buffer queue(s), then returns pointers to the playback buffers (left and right). If the audio is in mono mode, the R member of P is assigned NULL. If the buffers cannot advance because the ISR is using the next buffers in the queue, this function returns FALSE and pointers to the buffers before advancement are returned. For DMA playback mode, this function must be called, on average, at the same rate the ISR is playing buffers.</p> |
| <b>Example</b>       | <pre> AUDIO_PTR Pptr;  Audio_Install(AUDIO_PLAYBACK, AUDIO_PCM16, AUDIO_STEREO, 8.0,              16, 64, 8); Audio_FillBufs(0x0000); Audio_Enable();  /* loop forever */ while (1) {     TaskWaitSema(AudioSemaId);     <b>Audio_GetPlaybackBufs(&amp;Pptr);</b>      /* Pptr.L now points to the left playback audio buffer */     /* Pptr.R now points to the right playback audio buffer */      /* do audio processing here */ } </pre>   |

|                      |   |       |                          |             |   |               |         |           |                    |                 |    |         |               |          |               |            |                    |             |                    |                 |                             |                  |                             |                 |                             |                  |                             |                 |                              |                  |                              |                   |                    |                     |                              |            |        |               |         |                    |        |                    |        |                     |        |                     |        |                      |  |                      |                                       |                      |                     |                      |                     |
|----------------------|---|-------|--------------------------|-------------|---|---------------|---------|-----------|--------------------|-----------------|----|---------|---------------|----------|---------------|------------|--------------------|-------------|--------------------|-----------------|-----------------------------|------------------|-----------------------------|-----------------|-----------------------------|------------------|-----------------------------|-----------------|------------------------------|------------------|------------------------------|-------------------|--------------------|---------------------|------------------------------|------------|--------|---------------|---------|--------------------|--------|--------------------|--------|---------------------|--------|---------------------|--------|----------------------|--|----------------------|---------------------------------------|----------------------|---------------------|----------------------|---------------------|
| Function Name        | Audio_Init  |       |                          |             |   |               |         |           |                    |                 |    |         |               |          |               |            |                    |             |                    |                 |                             |                  |                             |                 |                             |                  |                             |                 |                              |                  |                              |                   |                    |                     |                              |            |        |               |         |                    |        |                    |        |                     |        |                     |        |                      |  |                      |                                       |                      |                     |                      |                     |
| Syntax               | BOOL Audio_Init();  |       |                          |             |   |               |         |           |                    |                 |    |         |               |          |               |            |                    |             |                    |                 |                             |                  |                             |                 |                             |                  |                             |                 |                              |                  |                              |                   |                    |                     |                              |            |        |               |         |                    |        |                    |        |                     |        |                     |        |                      |  |                      |                                       |                      |                     |                      |                     |
| Arguments            | None  |       |                          |             |   |               |         |           |                    |                 |    |         |               |          |               |            |                    |             |                    |                 |                             |                  |                             |                 |                             |                  |                             |                 |                              |                  |                              |                   |                    |                     |                              |            |        |               |         |                    |        |                    |        |                     |        |                     |        |                      |  |                      |                                       |                      |                     |                      |                     |
| Return Value         | BOOL  | TRUE  | Initialization succeeded |             |   |               |         |           |                    |                 |    |         |               |          |               |            |                    |             |                    |                 |                             |                  |                             |                 |                             |                  |                             |                 |                              |                  |                              |                   |                    |                     |                              |            |        |               |         |                    |        |                    |        |                     |        |                     |        |                      |  |                      |                                       |                      |                     |                      |                     |
|                      |   | FALSE | Initialization failed    |             |   |               |         |           |                    |                 |    |         |               |          |               |            |                    |             |                    |                 |                             |                  |                             |                 |                             |                  |                             |                 |                              |                  |                              |                   |                    |                     |                              |            |        |               |         |                    |        |                    |        |                     |        |                     |        |                      |  |                      |                                       |                      |                     |                      |                     |
| Description          | <p>This function initializes the audio codec and audio FIFO to a default state by resetting the FIFO circuitry. Upon reset, all previous codec and FIFO settings are lost. The codec will go through autocalibration twice during this call, which could require up to 800 sample periods.</p> <p><b>Default codec values:</b></p> <table><tr><td>data format</td><td>16-bit, PCM (pulse code modulation), mono</td></tr><tr><td>sampling rate</td><td>8.0 kHz</td></tr><tr><td>interface</td><td>Single-channel DMA</td></tr><tr><td>autocalibration</td><td>on</td></tr><tr><td>capture</td><td>PIO, disabled</td></tr><tr><td>playback</td><td>PIO, disabled</td></tr><tr><td>left input</td><td>LINE, +4.5 dB gain</td></tr><tr><td>right input</td><td>LINE, +4.5 dB gain</td></tr><tr><td>AUX1 left input</td><td>0.0 dB attenuation, mute on</td></tr><tr><td>AUX1 right input</td><td>0.0 dB attenuation, mute on</td></tr><tr><td>AUX2 left input</td><td>0.0 dB attenuation, mute on</td></tr><tr><td>AUX2 right input</td><td>0.0 dB attenuation, mute on</td></tr><tr><td>left DAC output</td><td>0.0 dB attenuation, mute off</td></tr><tr><td>right DAC output</td><td>0.0 dB attenuation, mute off</td></tr><tr><td>interrupt control</td><td>Interrupt disabled</td></tr><tr><td>digital mix control</td><td>0.0 dB attenuation, disabled</td></tr><tr><td>base count</td><td>0x0000</td></tr></table> <p><b>Default FIFO values:</b></p> <table><tr><td>DMA direction</td><td>Capture</td></tr><tr><td>capture AEF offset</td><td>0x0200</td></tr><tr><td>capture AFF offset</td><td>0x0200</td></tr><tr><td>playback AEF offset</td><td>0x0200</td></tr><tr><td>playback AFF offset</td><td>0x0200</td></tr><tr><td>flag pin A (CdcInt0)</td><td>0x0D → capture almost empty flag (AEF)</td></tr><tr><td>flag pin B (CdcInt1)</td><td>0x0F → capture almost full flag (AFF)</td></tr><tr><td>flag pin C (CdcErr0)</td><td>0x09 → playback AEF</td></tr><tr><td>flag pin D (CdcErr1)</td><td>0x0B → playback AFF</td></tr></table> |       |                          | data format | 16-bit, PCM (pulse code modulation), mono | sampling rate | 8.0 kHz | interface | Single-channel DMA | autocalibration | on | capture | PIO, disabled | playback | PIO, disabled | left input | LINE, +4.5 dB gain | right input | LINE, +4.5 dB gain | AUX1 left input | 0.0 dB attenuation, mute on | AUX1 right input | 0.0 dB attenuation, mute on | AUX2 left input | 0.0 dB attenuation, mute on | AUX2 right input | 0.0 dB attenuation, mute on | left DAC output | 0.0 dB attenuation, mute off | right DAC output | 0.0 dB attenuation, mute off | interrupt control | Interrupt disabled | digital mix control | 0.0 dB attenuation, disabled | base count | 0x0000 | DMA direction | Capture | capture AEF offset | 0x0200 | capture AFF offset | 0x0200 | playback AEF offset | 0x0200 | playback AFF offset | 0x0200 | flag pin A (CdcInt0) | 0x0D → capture almost empty flag (AEF) | flag pin B (CdcInt1) | 0x0F → capture almost full flag (AFF) | flag pin C (CdcErr0) | 0x09 → playback AEF | flag pin D (CdcErr1) | 0x0B → playback AFF |
| data format          | 16-bit, PCM (pulse code modulation), mono   |       |                          |             |   |               |         |           |                    |                 |    |         |               |          |               |            |                    |             |                    |                 |                             |                  |                             |                 |                             |                  |                             |                 |                              |                  |                              |                   |                    |                     |                              |            |        |               |         |                    |        |                    |        |                     |        |                     |        |                      |  |                      |                                       |                      |                     |                      |                     |
| sampling rate        | 8.0 kHz   |       |                          |             |   |               |         |           |                    |                 |    |         |               |          |               |            |                    |             |                    |                 |                             |                  |                             |                 |                             |                  |                             |                 |                              |                  |                              |                   |                    |                     |                              |            |        |               |         |                    |        |                    |        |                     |        |                     |        |                      |  |                      |                                       |                      |                     |                      |                     |
| interface            | Single-channel DMA  |       |                          |             |   |               |         |           |                    |                 |    |         |               |          |               |            |                    |             |                    |                 |                             |                  |                             |                 |                             |                  |                             |                 |                              |                  |                              |                   |                    |                     |                              |            |        |               |         |                    |        |                    |        |                     |        |                     |        |                      |  |                      |                                       |                      |                     |                      |                     |
| autocalibration      | on  |       |                          |             |   |               |         |           |                    |                 |    |         |               |          |               |            |                    |             |                    |                 |                             |                  |                             |                 |                             |                  |                             |                 |                              |                  |                              |                   |                    |                     |                              |            |        |               |         |                    |        |                    |        |                     |        |                     |        |                      |  |                      |                                       |                      |                     |                      |                     |
| capture              | PIO, disabled   |       |                          |             |   |               |         |           |                    |                 |    |         |               |          |               |            |                    |             |                    |                 |                             |                  |                             |                 |                             |                  |                             |                 |                              |                  |                              |                   |                    |                     |                              |            |        |               |         |                    |        |                    |        |                     |        |                     |        |                      |  |                      |                                       |                      |                     |                      |                     |
| playback             | PIO, disabled   |       |                          |             |   |               |         |           |                    |                 |    |         |               |          |               |            |                    |             |                    |                 |                             |                  |                             |                 |                             |                  |                             |                 |                              |                  |                              |                   |                    |                     |                              |            |        |               |         |                    |        |                    |        |                     |        |                     |        |                      |  |                      |                                       |                      |                     |                      |                     |
| left input           | LINE, +4.5 dB gain  |       |                          |             |   |               |         |           |                    |                 |    |         |               |          |               |            |                    |             |                    |                 |                             |                  |                             |                 |                             |                  |                             |                 |                              |                  |                              |                   |                    |                     |                              |            |        |               |         |                    |        |                    |        |                     |        |                     |        |                      |  |                      |                                       |                      |                     |                      |                     |
| right input          | LINE, +4.5 dB gain  |       |                          |             |   |               |         |           |                    |                 |    |         |               |          |               |            |                    |             |                    |                 |                             |                  |                             |                 |                             |                  |                             |                 |                              |                  |                              |                   |                    |                     |                              |            |        |               |         |                    |        |                    |        |                     |        |                     |        |                      |  |                      |                                       |                      |                     |                      |                     |
| AUX1 left input      | 0.0 dB attenuation, mute on   |       |                          |             |   |               |         |           |                    |                 |    |         |               |          |               |            |                    |             |                    |                 |                             |                  |                             |                 |                             |                  |                             |                 |                              |                  |                              |                   |                    |                     |                              |            |        |               |         |                    |        |                    |        |                     |        |                     |        |                      |  |                      |                                       |                      |                     |                      |                     |
| AUX1 right input     | 0.0 dB attenuation, mute on   |       |                          |             |   |               |         |           |                    |                 |    |         |               |          |               |            |                    |             |                    |                 |                             |                  |                             |                 |                             |                  |                             |                 |                              |                  |                              |                   |                    |                     |                              |            |        |               |         |                    |        |                    |        |                     |        |                     |        |                      |  |                      |                                       |                      |                     |                      |                     |
| AUX2 left input      | 0.0 dB attenuation, mute on   |       |                          |             |   |               |         |           |                    |                 |    |         |               |          |               |            |                    |             |                    |                 |                             |                  |                             |                 |                             |                  |                             |                 |                              |                  |                              |                   |                    |                     |                              |            |        |               |         |                    |        |                    |        |                     |        |                     |        |                      |  |                      |                                       |                      |                     |                      |                     |
| AUX2 right input     | 0.0 dB attenuation, mute on   |       |                          |             |   |               |         |           |                    |                 |    |         |               |          |               |            |                    |             |                    |                 |                             |                  |                             |                 |                             |                  |                             |                 |                              |                  |                              |                   |                    |                     |                              |            |        |               |         |                    |        |                    |        |                     |        |                     |        |                      |  |                      |                                       |                      |                     |                      |                     |
| left DAC output      | 0.0 dB attenuation, mute off  |       |                          |             |   |               |         |           |                    |                 |    |         |               |          |               |            |                    |             |                    |                 |                             |                  |                             |                 |                             |                  |                             |                 |                              |                  |                              |                   |                    |                     |                              |            |        |               |         |                    |        |                    |        |                     |        |                     |        |                      |  |                      |                                       |                      |                     |                      |                     |
| right DAC output     | 0.0 dB attenuation, mute off  |       |                          |             |   |               |         |           |                    |                 |    |         |               |          |               |            |                    |             |                    |                 |                             |                  |                             |                 |                             |                  |                             |                 |                              |                  |                              |                   |                    |                     |                              |            |        |               |         |                    |        |                    |        |                     |        |                     |        |                      |  |                      |                                       |                      |                     |                      |                     |
| interrupt control    | Interrupt disabled  |       |                          |             |   |               |         |           |                    |                 |    |         |               |          |               |            |                    |             |                    |                 |                             |                  |                             |                 |                             |                  |                             |                 |                              |                  |                              |                   |                    |                     |                              |            |        |               |         |                    |        |                    |        |                     |        |                     |        |                      |  |                      |                                       |                      |                     |                      |                     |
| digital mix control  | 0.0 dB attenuation, disabled  |       |                          |             |   |               |         |           |                    |                 |    |         |               |          |               |            |                    |             |                    |                 |                             |                  |                             |                 |                             |                  |                             |                 |                              |                  |                              |                   |                    |                     |                              |            |        |               |         |                    |        |                    |        |                     |        |                     |        |                      |  |                      |                                       |                      |                     |                      |                     |
| base count           | 0x0000  |       |                          |             |   |               |         |           |                    |                 |    |         |               |          |               |            |                    |             |                    |                 |                             |                  |                             |                 |                             |                  |                             |                 |                              |                  |                              |                   |                    |                     |                              |            |        |               |         |                    |        |                    |        |                     |        |                     |        |                      |  |                      |                                       |                      |                     |                      |                     |
| DMA direction        | Capture   |       |                          |             |   |               |         |           |                    |                 |    |         |               |          |               |            |                    |             |                    |                 |                             |                  |                             |                 |                             |                  |                             |                 |                              |                  |                              |                   |                    |                     |                              |            |        |               |         |                    |        |                    |        |                     |        |                     |        |                      |  |                      |                                       |                      |                     |                      |                     |
| capture AEF offset   | 0x0200  |       |                          |             |   |               |         |           |                    |                 |    |         |               |          |               |            |                    |             |                    |                 |                             |                  |                             |                 |                             |                  |                             |                 |                              |                  |                              |                   |                    |                     |                              |            |        |               |         |                    |        |                    |        |                     |        |                     |        |                      |  |                      |                                       |                      |                     |                      |                     |
| capture AFF offset   | 0x0200  |       |                          |             |   |               |         |           |                    |                 |    |         |               |          |               |            |                    |             |                    |                 |                             |                  |                             |                 |                             |                  |                             |                 |                              |                  |                              |                   |                    |                     |                              |            |        |               |         |                    |        |                    |        |                     |        |                     |        |                      |  |                      |                                       |                      |                     |                      |                     |
| playback AEF offset  | 0x0200  |       |                          |             |   |               |         |           |                    |                 |    |         |               |          |               |            |                    |             |                    |                 |                             |                  |                             |                 |                             |                  |                             |                 |                              |                  |                              |                   |                    |                     |                              |            |        |               |         |                    |        |                    |        |                     |        |                     |        |                      |  |                      |                                       |                      |                     |                      |                     |
| playback AFF offset  | 0x0200  |       |                          |             |   |               |         |           |                    |                 |    |         |               |          |               |            |                    |             |                    |                 |                             |                  |                             |                 |                             |                  |                             |                 |                              |                  |                              |                   |                    |                     |                              |            |        |               |         |                    |        |                    |        |                     |        |                     |        |                      |  |                      |                                       |                      |                     |                      |                     |
| flag pin A (CdcInt0) | 0x0D → capture almost empty flag (AEF)  |       |                          |             |   |               |         |           |                    |                 |    |         |               |          |               |            |                    |             |                    |                 |                             |                  |                             |                 |                             |                  |                             |                 |                              |                  |                              |                   |                    |                     |                              |            |        |               |         |                    |        |                    |        |                     |        |                     |        |                      |  |                      |                                       |                      |                     |                      |                     |
| flag pin B (CdcInt1) | 0x0F → capture almost full flag (AFF)   |       |                          |             |   |               |         |           |                    |                 |    |         |               |          |               |            |                    |             |                    |                 |                             |                  |                             |                 |                             |                  |                             |                 |                              |                  |                              |                   |                    |                     |                              |            |        |               |         |                    |        |                    |        |                     |        |                     |        |                      |  |                      |                                       |                      |                     |                      |                     |
| flag pin C (CdcErr0) | 0x09 → playback AEF   |       |                          |             |   |               |         |           |                    |                 |    |         |               |          |               |            |                    |             |                    |                 |                             |                  |                             |                 |                             |                  |                             |                 |                              |                  |                              |                   |                    |                     |                              |            |        |               |         |                    |        |                    |        |                     |        |                     |        |                      |  |                      |                                       |                      |                     |                      |                     |
| flag pin D (CdcErr1) | 0x0B → playback AFF   |       |                          |             |   |               |         |           |                    |                 |    |         |               |          |               |            |                    |             |                    |                 |                             |                  |                             |                 |                             |                  |                             |                 |                              |                  |                              |                   |                    |                     |                              |            |        |               |         |                    |        |                    |        |                     |        |                     |        |                      |  |                      |                                       |                      |                     |                      |                     |
| Example              | <pre>/* initialize audio hardware */ Audio_Init();</pre>  |       |                          |             |   |               |         |           |                    |                 |    |         |               |          |               |            |                    |             |                    |                 |                             |                  |                             |                 |                             |                  |                             |                 |                              |                  |                              |                   |                    |                     |                              |            |        |               |         |                    |        |                    |        |                     |        |                     |        |                      |  |                      |                                       |                      |                     |                      |                     |

This function performs the following actions:

- ☐ Disables audio
- ☐ Creates internal buffer queues
- ☐ Programs audio FIFO
- ☐ Programs audio codec
- ☐ Sets up events

**Example**

```
BOOL success;  
  
success = Audio_Install(AUDIO_CAPTURE, AUDIO_PCM16,  
    AUDIO_STEREO, 48.0, 8, 64, 4);  
  
if (success) {  
    /* do audio stuff */  
}
```



|                      |   |
|----------------------|---|
| <b>Function Name</b> | <b>Audio_InstallSema</b>  |
| <b>Syntax</b>        | long Audio_InstallSema(long <i>Semald</i> );  |
| <b>Arguments</b>     | long <i>Semald</i> ID of semaphore to install returned by TaskOpenSema()  |
| <b>Return Value</b>  | long              Old semaphore ID  |
| <b>Description</b>   | <p>This function installs a semaphore into the audio subsystem. The application must open the semaphore by calling TaskOpenSema(). Only AUDIO_CAPTURE and AUDIO_PLAYBACK modes use the semaphore. These modes use DMA and the audio FIFO that triggers an event. The ISR for this event manages the buffers and, when a new buffer is captured or played back, the semaphore is signaled. The semaphore allows an application to synchronize with the audio ISR.</p>                            |
| <b>Example</b>       | <pre>void AudioTask(void *p) {     BOOL success;     long AudioSemaId;     AUDIO_PTR P;      AudioSemaId = TaskOpenSema(-1,0);     <b>Audio_InstallSema(AudioSemaId);</b>     success = Audio_Install(AUDIO_CAPTURE, AUDIO_PCM16,         AUDIO_STEREO, 48.0, 8, 64, 4);      if (success) {         Audio_Enable();         while (1) {             TaskWaitSema(AudioSemaId);             Audio_GetCaptureBufs(&amp;P);             /* process captured audio here */         }     } }</pre> |

|                      |  |
|----------------------|--|
| <b>Function Name</b> | <b>Audio_PioIn</b>   |
| <b>Syntax</b>        | <code>void Audio_PioIn(void *left, void *right);</code>  |
| <b>Arguments</b>     | <div><div><code>void *left</code></div><div>Pointer to left sample storage</div></div> <div><div><code>void *right</code></div><div>Pointer to right sample storage</div></div>  |
| <b>Return Value</b>  | None   |
| <b>Description</b>   | <p>This function performs the following actions:</p> <ul style="list-style-type: none"><li><input type="checkbox"/> Inputs samples from the audio codec when the codec is programmed for programmed input/output (PIO) capture mode</li><li><input type="checkbox"/> Automatically detects the data format (8- or 16-bit, stereo or mono)</li><li><input type="checkbox"/> Waits until the codec is ready with samples before reading them</li></ul> <p>If the codec is in mono mode, only the left sample is captured. If the codec capture mode is not PIO or if capture is not enabled, the function returns immediately without waiting or reading from the codec.</p> |
| <b>Example</b>       | <pre>USHORT l,r;  /* ... other processing ... */ /* capture sample in PIO mode */ <b>Audio_PioIn((void*)&amp;l, (void*)&amp;r);</b> /* ... other processing ... */</pre>   |

|                      |  |
|----------------------|--|
| <b>Function Name</b> | <b>Audio_PioOut</b>  |
| <b>Syntax</b>        | <code>void Audio_PioOut(void *left, void *right);</code>   |
| <b>Arguments</b>     | <div><div><code>void *left</code></div><div>Pointer to left sample storage</div></div> <div><div><code>void *right</code></div><div>Pointer to right sample storage</div></div>  |
| <b>Return Value</b>  | None   |
| <b>Description</b>   | <p>This function performs the following actions:</p> <ul style="list-style-type: none"><li><input type="checkbox"/> Outputs samples from the audio codec when the codec is programmed for programmed input/output (PIO) playback mode</li><li><input type="checkbox"/> Automatically detects the data format (8- or 16-bit, stereo or mono)</li><li><input type="checkbox"/> Waits until the codec is ready for samples before writing them out</li></ul> <p>If the codec is in mono mode, only the left sample is used. If the codec playback mode is not PIO or if playback is not enabled, the function returns immediately without waiting or writing to the codec. This function does not modify the sample values.</p> |
| <b>Example</b>       | <pre>USHORT l,r;  /* ... other processing ... */ /* playback sample in PIO mode */ <b>Audio_PioOut((void*)&amp;l, (void*)&amp;r);</b> /* ... other processing ... */</pre>   |

**Function Name****Audio\_PioTest****Syntax**

```
void Audio_PioTest(BYTE Format, BOOL Stereo, float Fs, ULONG Ct);
```

**Arguments**

|                    |   |                              |
|--------------------|---|------------------------------|
| BYTE <i>Format</i> | AUDIO_PCM16                               | 16-bit PCM (signed)          |
|                    | AUDIO_PCM8                                | 8-bit PCM (unsigned)         |
|                    | AUDIO_ALAW8                               | 8-bit A-Law compression      |
|                    | AUDIO_ULAW8                               | 8-bit $\mu$ -Law compression |
| BOOL <i>Stereo</i> | TRUE                                      | Set up stereo mode           |
|                    | FALSE                                     | Set up mono mode             |
| float <i>Fs</i>    | Sampling frequency (kHz):                 |                              |
|                    | 5.5125                                    | 22.0500                      |
|                    | 6.6150                                    | 27.4286                      |
|                    | 8.0000                                    | 32.0000                      |
|                    | 9.6000                                    | 33.0750                      |
|                    | 11.0250                                   | 37.8000                      |
|                    | 16.0000                                   | 44.1000                      |
|                    | 18.9000                                   | 48.0000                      |
| ULONG <i>Ct</i>    | Number of loopback samples to run through |                              |

**Return Value**

None

**Description**

This function tests the audio subsystem by running in full-duplex loopback AUDIO\_PIO mode for *Ct* samples. If audio has already been installed using Audio\_Install(), it should be uninstalled using Audio\_UnInstall() before calling this function. Audio should not be enabled when this function is called.

**Example**

```
/* loopback 60 seconds of audio */  
Audio_PioTest(AUDIO_PCM16, AUDIO_STEREO, 48.0, 48000*60);
```

|                      |   |
|----------------------|---|
| <b>Function Name</b> | <b>Audio_PlaybackFromMemory</b>   |
| <b>Syntax</b>        | <code>void Audio_PlaybackFromMemory(AUDIO_PTR *P, ULONG NumBufs);</code>  |
| <b>Arguments</b>     | <p><code>AUDIO_PTR *P</code> Pointer to a buffer pointer structure whose elements point to preallocated buffers of memory that are filled with audio playback data</p> <p><code>ULONG NumBufs</code> Amount of audio to play back; value specified as an integral number of DMA buffers as set up by <code>Audio_Install()</code></p>   |
| <b>Return Value</b>  | None  |
| <b>Description</b>   | <p>This function plays back a specified amount of audio data from buffers pointed to by the members of P. Before calling <code>Audio_PlaybackFromMemory()</code>, you must first call <code>Audio_Install()</code> to set up the audio hardware for DMA playback. The <code>NumBufs</code> argument specifies how much audio to play in number of buffers. The buffer size is determined when you call <code>Audio_Install()</code>. If the audio is set up for mono mode, the L member of P is ignored.</p> <hr/> <p><b>Note:</b></p> <ol style="list-style-type: none"> <li>1) Do not call <code>Audio_PlaybackFromMemory()</code> while audio is enabled.</li> <li>2) <code>Audio_PlaybackFromMemory()</code> must be called from a task other than the default task because it waits on a semaphore.</li> </ol> <hr/>   |
| <b>Example</b>       | <pre> void AudioTask(void *P) {     AUDIO_PTR Pptr;     AUDIO_MET AM;     BYTE  Format   = AUDIO_PCM16;     BOOL  Stereo   = AUDIO_STEREO;     float Fs       = 48.0;     BYTE  BlockCt  = 20;     BYTE  BlockSz  = 50;     BYTE  BuffCt   = 8;     ULONG NumBufs = Fs*10;      /* dummy install to set audio metrics */     Audio_Install(AUDIO_PLAYBACK, Format, Stereo, Fs, BlockCt,                  BlockSz, BuffCt);      /* get the metrics of the new settings */     Audio_GetMetrics(&amp;AM);      /* allocate some DRAM storage for the playback audio */     Pptr.L = (void*)memalign(64, AM.ByteSz*NumBufs);     if (Stereo)         Pptr.R = (void*)memalign(64, AM.ByteSz*NumBufs);     /* fill up the allocated playback buffers here */     /* playback some audio from the allocated buffers */     Audio_PlaybackFromMemory(&amp;Pptr, NumBufs); </pre> |

```
    /* uninstall the audio settings */
    Audio_UnInstall();

    /* free up the buffer storage */
    free((void*)Pptr.L);
    if (Stereo)
        free((void*)Pptr.R);
}
```

|   |   |  |                            |
|---|---|--|----------------------------|
| Function Name   | Audio_ProgramAux1   |  |                            |
| Syntax  | void Audio_ProgramAux1(BOOL <i>lmute</i> , BOOL <i>rmute</i> , float <i>lg</i> , float <i>rg</i> );   |  |                            |
| Arguments   | BOOL <i>lmute</i>   | AUDIO_MUTE   | Left auxiliary 1 mute on   |
|   |   | AUDIO_UNMUTE   | Left auxiliary 1 mute off  |
|   | BOOL <i>rmute</i>   | AUDIO_MUTE   | Right auxiliary 1 mute on  |
|   |   | AUDIO_UNMUTE   | Right auxiliary 1 mute off |
|   | float <i>lg</i>   | Left auxiliary 1 gain in dB<br>-34.5 dB <= lg <= +12.0 dB (in 1.5 dB steps)  |                            |
|   | float <i>rg</i>   | Right auxiliary 1 gain in dB<br>-34.5 dB <= rg <= +12.0 dB (in 1.5 dB steps) |                            |
| Return Value  | None  |  |                            |
| Description   | <p>This function sets up auxiliary 1 analog mixing. The aux1 inputs are mixed in the analog domain with the outputs of the digital-to-analog converters (DACs). These mixed signals appear on the codec analog outputs. If mute is turned on, no mixing takes place. Otherwise, the aux1 inputs are run through a gain stage and then mixed. The codec does not go through autocalibration.</p> |  |                            |
| <div><b>Note:</b><p>This function does <i>not</i> affect the aux1 inputs to the analog-to-digital converters (ADCs). You must program the input gains to the ADCs by using Audio_ProgramInputs().</p></div> |   |  |                            |
| Example   | <pre>/* unmute aux1 inputs and set +3.0 dB gain */ Audio_ProgramAux1(AUDIO_UNMUTE, AUDIO_UNMUTE, +3.0, +3.0);  /* unmute aux1 inputs and set -3.0 dB gain */ /* which is +3.0 dB attenuation */ Audio_ProgramAux1(AUDIO_UNMUTE, AUDIO_UNMUTE, -3.0, -3.0);</pre>  |  |                            |

|               |   |   |                    |
|---------------|---|---|--------------------|
| Function Name | Audio_ProgramDacs   |   |                    |
| Syntax        | void Audio_ProgramDacs(BOOL <i>lmute</i> , BOOL <i>rmute</i> , float <i>lg</i> , float <i>rg</i> );   |   |                    |
| Arguments     | BOOL <i>lmute</i>   | AUDIO_MUTE  | Left DAC mute on   |
|               |   | AUDIO_UNMUTE  | Left DAC mute off  |
|               | BOOL <i>rmute</i>   | AUDIO_MUTE  | Right DAC mute on  |
|               |   | AUDIO_UNMUTE  | Right DAC mute off |
|               | float <i>lg</i>   | Left DAC gain in dB<br>-94.5 dB <= <i>lg</i> <= 0.0 dB (in 1.5 dB steps)  |                    |
|               | float <i>rg</i>   | Right DAC gain in dB<br>-94.5 dB <= <i>rg</i> <= 0.0 dB (in 1.5 dB steps) |                    |
| Return Value  | None  |   |                    |
| Description   | This function sets up the digital-to-analog converter (DAC) outputs of the audio codec. The codec does not go through autocalibration.  |   |                    |
| Example       | <pre>/* unmute both codec DACs and set 0 dB gain */ Audio_ProgramDacs(AUDIO_UNMUTE, AUDIO_UNMUTE, 0.0, 0.0);  /* unmute both codec DACs and set -9.0 dB gain */ /* which is +9.0 dB attenuation */ Audio_ProgramDacs(AUDIO_UNMUTE, AUDIO_UNMUTE, -9.0, -9.0);</pre> |   |                    |



|               |  |   |                      |
|---------------|--|---|----------------------|
| Function Name | Audio_ProgramDigitalMix  |   |                      |
| Syntax        | void Audio_ProgramDigitalMix(BOOL <i>enable</i> , float <i>gain</i> );   |   |                      |
| Arguments     | BOOL <i>enable</i>   | TRUE  | Digital mix enabled  |
|               |  | FALSE   | Digital mix disabled |
|               | float <i>gain</i>  | ADC gain to DAC in dB<br>-94.5 dB <= gain <= 0.0 dB (in 1.5 dB steps) |                      |
| Return Value  | None   |   |                      |
| Description   | This function sets up the digital mix capabilities of the audio codec. With digital mixing enabled, the output of the analog-to-digital converters (ADCs) is run through a gain stage and then digitally mixed with the input to the digital-to-analog converters (DACs). The codec does not go through autocalibration. |   |                      |
| Example       | <pre>/* enable digital mixing and set -6.0 dB gain */<br/>Audio_ProgramDigitalMix(TRUE, -6.0);</pre>   |   |                      |

|               |   |   |                                       |
|---------------|---|---|---------------------------------------|
| Function Name | Audio_ProgramInputs   |   |                                       |
| Syntax        | void Audio_ProgramInputs(BYTE <i>lsrc</i> , BYTE <i>rsrc</i> , float <i>lg</i> , float <i>rg</i> );                       |   |                                       |
| Arguments     | BYTE <i>lsrc</i>  | Selects which input source to use for the left input to the audio codec:                                |                                       |
|               |   | AUDIO_LINE  | Select line input                     |
|               |   | AUDIO_AUX1  | Select auxiliary 1 input              |
|               |   | AUDIO_MIXED   | Select line input with post-mixed DAC |
|               | BYTE <i>rsrc</i>  | Selects which input source to use for the right input to the audio codec:                               |                                       |
|               |   | AUDIO_LINE  | Select line input                     |
|               |   | AUDIO_AUX1  | Select auxiliary 1 input              |
|               |   | AUDIO_MIXED   | Select line input with post-mixed DAC |
|               | float <i>lg</i>   | Sets the audio codec's left input gain in decibels<br>0.00 dB <= <i>lg</i> <= +22.5 dB (+1.5 dB steps)  |                                       |
|               | float <i>rg</i>   | Sets the audio codec's right input gain in decibels<br>0.00 dB <= <i>rg</i> <= +22.5 dB (+1.5 dB steps) |                                       |
| Return Value  | None  |   |                                       |
| Description   | This function programs the inputs to the audio codec.   |   |                                       |
| Example       | <pre>/* program both inputs to LINE with +4.5 dB gain */<br/>Audio_ProgramInputs(AUDIO_LINE, AUDIO_LINE, 4.5, 4.5);</pre> |   |                                       |

|               |   |   |                              |
|---------------|---|---|------------------------------|
| Function Name | Audio_SetBufferIndexes  |   |                              |
| Syntax        | BOOL Audio_SetBufferIndexes(BYTE <i>Capp</i> , BYTE <i>Cisr</i> , BYTE <i>Papp</i> , BYTE <i>Pisr</i> );  |   |                              |
| Arguments     | BYTE <i>Capp</i>  | Application index into capture buffer queue(s)  |                              |
|               | BYTE <i>Cisr</i>  | ISR index into capture buffer queue(s)          |                              |
|               | BYTE <i>Papp</i>  | Application index into playback buffer queue(s) |                              |
|               | BYTE <i>Pisr</i>  | ISR index into playback buffer queue(s)         |                              |
| Return Value  | BOOL  | TRUE  | Success                      |
|               |   | FALSE   | Failure, indexes are invalid |
| Description   | <p>This function sets the internal indexes into the buffer queue(s) and allows the application to specify custom starting points for the indexes. Before calling this function, you must first call <code>Audio_Install()</code>. Remember that <i>Cisr</i> cannot equal <i>Capp</i>, and <i>Pisr</i> cannot equal <i>Papp</i>; if these indexes are equal, this function returns FALSE. Also, none of the indexes can exceed or equal the number of buffers in the queue as specified in the call to <code>Audio_Install()</code>. All four indexes must be specified, even though only one set (either capture or playback) is used. See Section 3.2, <i>Audio Buffering</i>, for more information regarding the buffering structure.</p> |   |                              |
|               | <div><b>Note:</b><ol style="list-style-type: none"><li>1) Do not call <code>Audio_SetBufferIndexes()</code> while audio is enabled.</li><li>2) This function normally is not needed unless the application requires custom settings.</li></ol></div>  |   |                              |
| Example       | <pre>Audio_Install(AUDIO_PLAYBACK, AUDIO_PCM8, AUDIO_MONO, 16.0, 8,               100, 6); Audio_SetBufferIndexes(1,5,1,5);</pre>   |   |                              |

**Function Name**

**Audio\_SetSampleRate**

**Syntax**

void Audio\_SetSampleRate(float Fs);

**Arguments**

| float Fs | Desired sampling rate in kHz: |
|----------|-------------------------------|
| 5.5125   | 22.05                         |
| 6.615    | 27.4286                       |
| 8.0      | 32.0                          |
| 9.6      | 33.075                        |
| 11.025   | 37.8                          |
| 16.0     | 44.1                          |
| 18.9     | 48.0                          |

**Return Value**

None

**Description**

This function sets the sample rate of the audio codec. The codec must go through autocalibration when the sample rate is changed, which could take up to 400 sample periods.

**Example**

```
/* set 8.0kHz sampling rate */  
Audio_SetSampleRate(8.0);
```

|                      |  |
|----------------------|--|
| <b>Function Name</b> | <b>Audio_UnInstall</b>   |
| <b>Syntax</b>        | <code>void Audio_UnInstall();</code>   |
| <b>Arguments</b>     | None   |
| <b>Return Value</b>  | None   |
| <b>Description</b>   | <p>This functions performs the following actions:</p> <ul style="list-style-type: none"><li><input type="checkbox"/> Uninstalls settings set up by Audio_Install()</li><li><input type="checkbox"/> Disables audio</li><li><input type="checkbox"/> Frees up buffer queue storage</li><li><input type="checkbox"/> Disables all audio events</li><li><input type="checkbox"/> Disables audio codec</li></ul> |
| <b>Example</b>       | <code>Audio_UnInstall();</code>  |

# Video Display API

---

---

---

This chapter discusses the video display macros and data types. It also describes, in alphabetical order, the application programming interface (API) functions associated with the video display driver for the TMS320C8x software development board (SDB).

| <b>Topic</b>   | <b>Page</b> |
|--|-------------|
| <b>4.1 Video Display API Macros and Data Types .....</b> | <b>4-2</b>  |
| <b>4.2 Video Overlay .....</b>                           | <b>4-8</b>  |
| <b>4.3 Video Display Window .....</b>                    | <b>4-9</b>  |
| <b>4.4 Video Display API Functions .....</b>             | <b>4-10</b> |

## 4.1 Video Display API Macros and Data Types

Table 2–1 describes the macros used by the video display API and lists the API functions that use each macro. Figure 4–1 provides definitions for the video display API data types. These macros and data types, as well as the API function prototypes, are defined in <display.h>. The object code resides in sdbdrvs.lib.

*Table 4–1. Video Display API Macros*

(a) SDB pixel formats

| Macro Name            | Lookup Key | Description of Pixel Bits   | Data Bits | Overlay Bits |
|-----------------------|------------|---|-----------|--------------|
| #define DISPLAY_P8    | s19        | 8-bit pseudocolor:<br>ddddddd   | 8         | 0            |
| #define DISPLAY_DXRGB | d2         | 32-bit direct color graphics with overlay field:<br>xxxxxxxxrrrrrrrrggggggbbbbbbb | 24        | 8            |
| #define DISPLAY_DBGRX | d4         | 32-bit direct color graphics with overlay field:<br>bbbbbbggggggrrrrrrxxxxxxxx    | 24        | 8            |
| #define DISPLAY_D565  | d7         | 16-bit direct color graphics:<br>rrrrggggggbbbb                                   | 16        | 0            |
| #define DISPLAY_D555  | d10        | 16-bit direct color graphics:<br>rrrrggggggbbbb                                   | 15        | 1            |
| #define DISPLAY_D664  | d13        | 16-bit direct color graphics:<br>rrrrggggggbbbb                                   | 16        | 0            |
| #define DISPLAY_D444  | d16        | 16-bit direct color graphics with overlay field:<br>rrrgggbbbbbxxx                | 12        | 4            |
| #define DISPLAY_TXRGB | t2         | 32-bit true color graphics:<br>xxxxxxxxrrrrrrrrggggggbbbbbbb                      | 24        | 0            |
| #define DISPLAY_TBGRX | t4         | 32-bit true color graphics:<br>bbbbbbggggggrrrrrrxxxxxxxx                         | 24        | 0            |
| #define DISPLAY_T565  | t7         | 16-bit true color graphics:<br>rrrrggggggbbbb                                     | 16        | 0            |
| #define DISPLAY_T555  | t10        | 16-bit true color graphics:<br>rrrrggggggbbbb                                     | 15        | 0            |
| #define DISPLAY_T664  | t13        | 16-bit true color graphics:<br>rrrrggggggbbbb                                     | 16        | 0            |
| #define DISPLAY_T444  | t16        | 16-bit true color graphics:<br>rrrgggbbbbbxxxx                                    | 12        | 0            |

**Legend:** d data bits                      b bits of blue field  
               g bits of green field        x bits of do not care field  
               r bits of red field

**Note:** These macros are used by the function Display\_SetMode().

Table 2–1. Video Display API Macros (Continued)

## (b) Output modes of the SDB graphics

| Macro Name                  | Value | Description   |
|-----------------------------|-------|---|
| #define DISPLAY_PASSTHROUGH | 0x01  | Set the graphics output to VGA pass-through. The pass-through cable must be connected for this mode.  |
| #define DISPLAY_VIDEO       | 0x02  | Set the graphics output to video mode (that is, the output of the RAMDAC).  |
| #define DISPLAY_OVERLAY     | 0x03  | Set the graphics output to mixed overlay mode. The input from the VGA pass-through cable is mixed with the RAMDAC output to form video overlaid onto VGA. The pass-through cable must be connected for this mode. |

**Notes:** 1) For information on how to connect the VGA pass-through cable, refer to the *TMS320C8x Software Development Board Installation Guide*.

2) These macros are used by the function `Display_SetMode()`.

## (c) Display buffer identifiers

| Macro Name               | Value | Description             |
|--------------------------|-------|-------------------------|
| #define DISPLAY_INACTIVE | 0x01  | Inactive display buffer |
| #define DISPLAY_ACTIVE   | 0x02  | Active display buffer   |
| #define DISPLAY_BUFF1    | 0x03  | Display buffer number 1 |
| #define DISPLAY_BUFF2    | 0x04  | Display buffer number 2 |

**Note:** These macros are used by the following functions:  
`Display_GetBuffer()`  
`Display_SetPixel()`



**Figure 4–1. Video Display API Data Types***(a) Monitor timing parameters for the 'C80 frame timer controller*

```
typedef struct {  
    char    active;    /* Active: 0=don't use, 1=can use, -1=end of table    */  
    USHORT Rh;        /* Horizontal resolution (in pixels)                */  
    USHORT Rv;        /* Vertical resolution (in pixels)                  */  
    float Fv;        /* Vertical frequency, refresh rate (in Hz)          */  
    float Fh;        /* Horizontal frequency, line rate (in kHz)          */  
    float Fd;        /* Pixel frequency, dot clock (in MHz)              */  
    float Ths;        /* Horizontal sync interval (in  $\mu$ s)                */  
    float Thbp;       /* Horizontal back porch interval (in  $\mu$ s)           */  
    float Tvs;        /* Vertical sync interval (in  $\mu$ s)                  */  
    float Tvbp;       /* Vertical back porch interval (in  $\mu$ s)             */  
    BYTE Sh;         /* Horizontal sync polarity: 0=-, 1=+                */  
    BYTE Sv;         /* Vertical sync polarity: 0=-, 1=+                */  
  
} DISPLAY_MT;
```

*(b) Type definition of DISPLAY\_MET*

```
typedef struct {  
    BYTE    FtNumber;    /* 'C80 frame timer used: 0 or 1                    */  
    BYTE    FclkRatio;   /* DCLK/FCLK ratio (default = 8)                    */  
    BYTE    RclkRatio;   /* DCLK/RCLK ratio                                  */  
    BYTE    BPP;        /* Bits per pixel: 8, 16, or 32                      */  
    USHORT Rh;          /* Horizontal resolution (in pixels)                  */  
    USHORT Rv;          /* Vertical resolution (in pixels)                    */  
    ULONG Vram;         /* Address of the start of VRAM                      */  
    ULONG VramLen;      /* Length of VRAM (in bytes)                         */  
    ULONG Buff1;        /* Frame buffer 1 address                            */  
    ULONG Buff2;        /* Frame buffer 2 address                            */  
    ULONG Pitch;        /* Frame buffer pitch (in bytes)                      */  
    ULONG CurBuff;      /* Current frame buffer address                      */  
    USHORT x;           /* X-coordinate of display window (in pixels)        */  
    USHORT y;           /* Y-coordinate of display window (in pixels)        */  
    USHORT dx;          /* Width of display window (in pixels)               */  
    USHORT dy;          /* Height of display window (in pixels)              */  
    float Fv;          /* Vertical frequency, refresh rate (in Hz)           */  
    float Fh;          /* Horizontal frequency, line rate (in kHz)           */  
    float Fd;          /* Pixel frequency, dot clock (in MHz)               */  
    float Ths;          /* Horizontal sync interval (in  $\mu$ s)                  */  
    float Thbp;         /* Horizontal back porch interval (in  $\mu$ s)            */  
    float Tvs;          /* Vertical sync interval (in  $\mu$ s)                   */  
    float Tvbp;         /* Vertical back porch interval (in  $\mu$ s)              */  
  
} DISPLAY_MET;
```

### 4.1.1 Color Modes

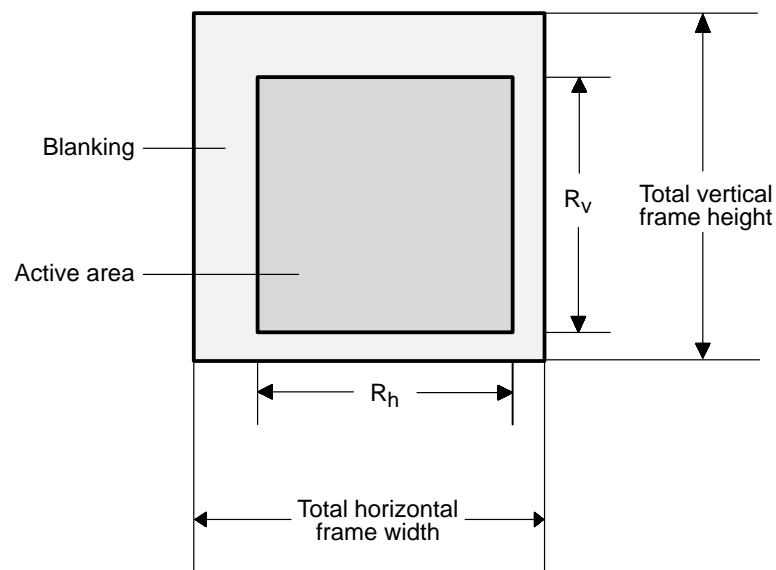
The pixel bit descriptions in Table 2–1(a) refer to three color modes: pseudo, direct, and true. Pseudocolor mode uses an 8-bit color value as a lookup index into the three color palette RAMs (one each for red, green, and blue) of the display RAMDAC. Pseudocolor mode is sometimes referred to as grayscale color. True color mode is similar to pseudocolor mode except three separate indexes are extracted from the color value and are used to index the color palette RAMs, one each for the red, green and blue. Direct color mode extracts the red, green, and blue indexes from the color value, bypasses the color palette RAMs, and outputs these values directly to the color digital-to-analog converters (DACs).

### 4.1.2 Monitor Timing Parameters

The structure type definition in Figure 4–1(a) defines the monitor timing parameters for programming the frame timer controller in the 'C80. Generally, you define a table of such structures with one entry for each display mode.

The parameters defined in the MT (monitor timing) structure derive the timing register values of the 'C80 frame timer controller. Figure 4–2 shows the layout of a typical video frame. All of the graphic modes defined in this API are *non-interlaced*; that is, each frame consists of a single vertical field and all of the lines in the frame are scanned out sequentially, one right after the other. This method of scanning is also called a progressive scan.

Figure 4–2. Relationship Between the Display Resolution and the Video Frame



A typical value of  $R_h$  is 640 pixels and a typical value of  $R_v$  is 480 pixels. Generally, the horizontal active area is about 76% of the total horizontal frame width, and the vertical active area is about 94% of the total vertical frame height.

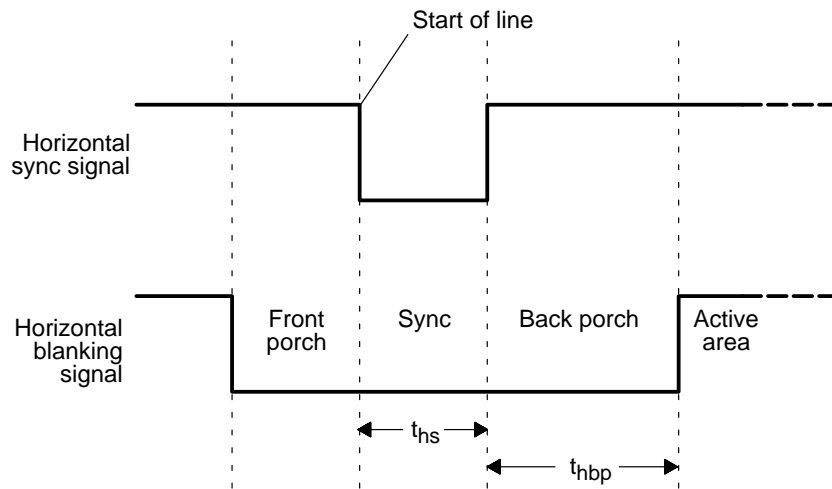
Because the MT structure does not define the total frame size, you must derive it by calculating pixels per line for the total horizontal frame width and lines per frame for the total vertical frame height. These calculations assume MHz frequencies:

$$\text{Total horizontal frame width} = \frac{F_d \text{ (million pixels/s)}}{F_h \text{ (million lines/s)}} = \frac{F_d}{F_h} \text{ (pixels/line)}$$

$$\text{Total vertical frame height} = \frac{F_h \text{ (million pixels/s)}}{F_v \text{ (million frames/s)}} = \frac{F_h}{F_v} \text{ (lines/frame)}$$

Figure 4–3 illustrates the horizontal sync and porch times in relation to the frame. The intervals  $t_{hs}$  and  $t_{hbp}$  correspond to the sync and back porch intervals of the horizontal sync and blanking signals. Similarly,  $t_{vs}$  and  $t_{vbp}$  have the same relationship to the vertical sync and blanking signals. For more details on the video timing signals, refer to the *TMS320C80 (MVP) Video Controller User's Guide*.

Figure 4–3. Horizontal Sync and Porch Times of the Video Frame



The frame timer registers derived from the data in the MT structure for both horizontal and vertical signals are:

- ☐ HESYNC (horizontal end sync)
- ☐ HEBLNK (horizontal end blank)
- ☐ HSBLNK (horizontal start blank)
- ☐ VESYNC (vertical end sync)
- ☐ VEBLNK (vertical end blank)
- ☐ VSBLNK (vertical start blank)

The following calculations are generalized. In the actual code, rounding is done and the horizontal registers are adjusted for the FCLK (frame clock) ratio. These calculations assume MHz frequencies and time intervals in  $\mu$ s times:

$$\begin{aligned}\text{HESYNC} &= t_{hs} * F_d \text{ (pixels)} \\ \text{HEBLNK} &= (t_{hs} + t_{hbp}) * F_d \text{ (pixels)} \\ \text{HSBLNK} &= (t_{hs} + t_{hbp}) * F_d + R_h \text{ (pixels)} \\ \text{VESYNC} &= t_{vs} + F_h \text{ (lines)} \\ \text{VEBLNK} &= (t_{vs} + t_{vbp}) * F_h \text{ (lines)} \\ \text{VSBLNK} &= (t_{vs} + t_{vbp}) * F_h + R_v \text{ (lines)}\end{aligned}$$

Remember that you use a table of MT structures to obtain the monitor timing parameters. You supply  $R_h$ ,  $R_v$ , and  $F_v$ ; then the software uses these values to find a match in the table. Once a match is found,  $F_h$ ,  $F_d$ ,  $t_{hs}$ ,  $t_{hbp}$ ,  $t_{vs}$ , and  $t_{vbp}$  are read from the table.

To create a custom table, copy the default table to your own code, modify it as needed, and pass a pointer to it to the `Display_InstallTimingTable()` function. Then all timing parameters will be read from this custom table. Example 4–1 illustrates an MT table of custom timing parameters. The values shown in Example 4–1 are the default values, but the structure array `MyTimingTable` is a separate entity from the default table. Once you create your custom table, you modify the default values.

**Example 4–1. Sample MT Table of Custom Timing Parameters**

```
static DISPLAY_MT MyTimingTable[] = {
/*   A      Rh      Rv      Fv      Fh      Fd      Ths      Thbp      Tvs      Tvbp      Sh      Sv  */
/*   pels   pels   Hz      kHz      MHz      μs      μs      μs      μs      0      0  */
{ 1, 640, 480, 60.0, 31.4, 25.2, 1.00, 2.00, 100.0, 600.0, 0, 0 },
{ 1, 640, 480, 72.0, 37.7, 31.2, 1.00, 3.00, 100.0, 600.0, 0, 0 },
{ 1, 800, 600, 60.0, 37.8, 40.0, 1.00, 2.00, 100.0, 400.0, 0, 0 },
{ 1, 1024, 768, 60.0, 48.3, 65.0, 1.00, 2.75, 100.0, 350.0, 0, 0 },
{ 1, 1024, 768, 70.0, 56.4, 75.0, 1.00, 2.00, 100.0, 350.0, 0, 0 },
{ 1, 1280, 1024, 60.0, 63.9, 114.0, 1.00, 2.40, 100.0, 400.0, 0, 0 },
{ 1, 1600, 1200, 60.0, 76.2, 156.0, 1.00, 2.00, 100.0, 600.0, 0, 0 },
{ -1, 0000, 0000, 00.0, 00.0, 000.0, 0.00, 0.00, 000.0, 000.0, 0, 0 }
};
```

In the application code, call `Display_InstallTimingTable(MyTimingTable)` to define your custom table.

**4.1.3 Metric Parameters**

The type definition for `DISPLAY_MET` in Figure 4–1(b) defines metric parameters of the current display state. Define a variable of this type, then pass a pointer to it as an argument to `Display_GetMetrics()`. `Display_GetMetrics()` fills in the structure members.

**4.2 Video Overlay**

Use the `Display_SetOverlayParams()` function to set up the video overlay feature of the SDB. Because of the way video overlay is achieved on the SDB, it is important that the 'C80-RAMDAC timing match with the timing of the VGA pass-through input. Generally, the VGA input is the VGA output of the graphics card that is currently installed in your PC. The timing of one VGA card differs from another VGA card, so you must adjust the overlay timing for each card that you use. Some trial and error may be necessary to get the right settings.

### 4.3 Video Display Window

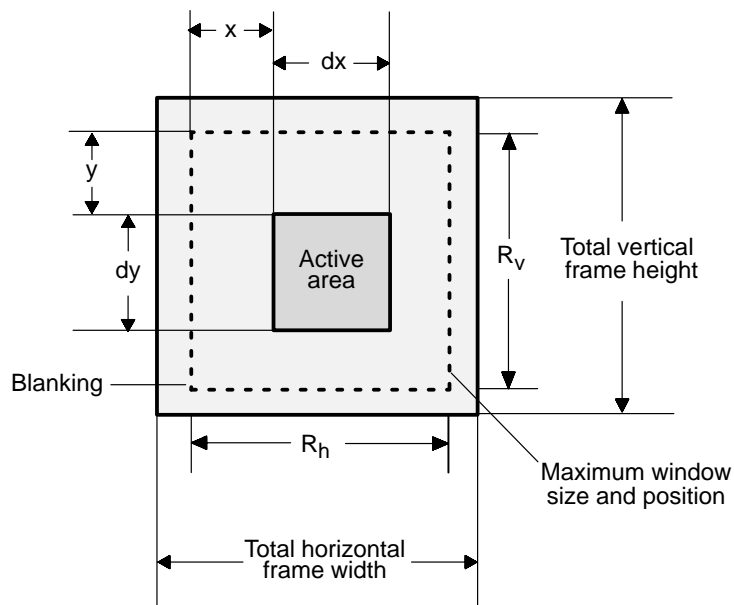
The API supplies functions for setting up a display window by calculating the timing parameters for full resolution and then adjusting the blanking parameters to produce the window. The default is a window with the same dimensions as the resolution of the display. Figure 4–4 illustrates the window feature. Serial register transfer (SRT) cycles are generated only during active time and not during blanking. This means that pixels are shifted out of the VRAMs only when the raster beam is within the specified window. The active area is defined by blanking.

#### Unsupported Resolutions Could Damage Your Monitor

Some monitors do not support the higher resolutions. Check your monitor specifications before attempting to drive it at a high resolution. Some monitors do not support resolutions greater than  $1024 \times 768$ . Remember also to verify the supported refresh rates.

If you have only 2M bytes of VRAM, you cannot set up a  $1024 \times 768$  display with 32 BPP (bits per pixel) because that would require  $1024 \times 768 \times 4 = 3\,145\,728$  bytes, which is more than 2M bytes. You can, however, set up a window of  $512 \times 512$  at 32 BPP within the  $1024 \times 768$  display. This would require only  $512 \times 512 \times 4 = 1\,048\,576$  bytes, which is much less than 2M bytes.

Figure 4–4. Video Display Driver Window Feature



## 4.4 Video Display API Functions

Listed below in alphabetical order are the video display API functions. Use this list as a table of contents to the video display API functions.

| Function                         | Page | Function                       | Page |
|----------------------------------|------|--------------------------------|------|
| Display_Disable .....            | 4-11 | Display_SetMode .....          | 4-25 |
| Display_Enable .....             | 4-12 | Display_SetOverlayParams ...   | 4-27 |
| Display_FillBufs .....           | 4-13 | Display_SetPaletteAddress ...  | 4-29 |
| Display_GetBuffer .....          | 4-14 | Display_SetPitch .....         | 4-30 |
| Display_GetMetrics .....         | 4-15 | Display_SetPixel .....         | 4-31 |
| Display_GetTvpRegs .....         | 4-16 | Display_SetSyncPolarities .... | 4-32 |
| Display_Init .....               | 4-17 | Display_SetVgaPalette .....    | 4-33 |
| Display_InstallSema .....        | 4-18 | Display_SetWindow .....        | 4-34 |
| Display_InstallTimingTable ..... | 4-19 | Display_ToggleBuffers .....    | 4-35 |
| Display_MoveWindow .....         | 4-20 | Display_TvpRegIn .....         | 4-36 |
| Display_ReadPalette .....        | 4-21 | Display_TvpRegOut .....        | 4-37 |
| Display_SetBufferAddresses ....  | 4-22 | Display_WaitEndOfFrame ....    | 4-38 |
| Display_SetDotClock .....        | 4-23 | Display_WritePalette .....     | 4-39 |
| Display_SetGreyScalePalette ...  | 4-24 |                                |      |

|                      |  |
|----------------------|--|
| <b>Function Name</b> | <b>Display_Disable</b>   |
| <b>Syntax</b>        | <code>void Display_Disable();</code>   |
| <b>Arguments</b>     | None   |
| <b>Return Value</b>  | None   |
| <b>Description</b>   | This function disables the display hardware by turning off the frame timer controller and the serial-data clock inputs to the 'C80.  |
| <b>Example</b>       | <pre>BOOL success;  Display_Init();  success = Display_SetMode(640, 480, 60, Display_P8,     DISPLAY_VIDEO);  if (success) {     Display_Enable();     /* ... do main programming ... */ }  Display_Disable();</pre> |



|               |   |
|---------------|---|
| Function Name | <b>Display_Enable</b>   |
| Syntax        | void Display_Enable();  |
| Arguments     | None  |
| Return Value  | None  |
| Description   | <p>This function enables the display hardware by turning on the frame timer controller and the serial-data clock inputs to the 'C80.</p> <p>When you call Display_SetMode(), the display hardware is disabled, so you must call Display_Enable() after Display_SetMode() to enable the display hardware. This allows you to make other display adjustments after calling Display_SetMode() and before enabling the display. Making adjustments while the display is disabled eliminates flicker. For example, you would call Display_SetWindow() <i>after</i> calling Display_SetMode() but <i>before</i> calling Display_Enable().</p> |
| Example       | <pre>BOOL success;  Display_Init();  success = Display_SetMode(640, 480, 60, DISPLAY_P8,     DISPLAY_VIDEO);  if (success) {     <b>Display_Enable();</b>     /* ... do main programming ... */ }  Display_Disable();</pre>   |

|                      |   |
|----------------------|---|
| <b>Function Name</b> | <b>Display_FillBufs</b>   |
| <b>Syntax</b>        | void Display_FillBufs(ULONG <i>val</i> );   |
| <b>Arguments</b>     | ULONG <i>val</i> Value to fill display buffers with   |
| <b>Return Value</b>  | None  |
| <b>Description</b>   | This function fills both display buffers with the value specified. The unit of storage is a ULONG. Therefore, if the display is 16 BPP, the <i>val</i> argument must have the fill value in both the upper and lower 16 bits. |
| <b>Example</b>       | <pre>Display_SetMode(640,480,60,DISPLAY_T555,DISPLAY_VIDEO); Display_FillBufs(0x001F001F); /* fill with blue */</pre>   |

|                      |  |                                 |   |
|----------------------|--|---------------------------------|---|
| <b>Function Name</b> | <b>Display_GetBuffer</b>   |                                 |   |
| <b>Syntax</b>        | ULONG Display_GetBuffer(BYTE <i>buffid</i> );  |                                 |   |
| <b>Arguments</b>     | BYTE <i>buffid</i>   | DISPLAY_INACTIVE                | Return address of inactive display buffer |
|                      |  | DISPLAY_ACTIVE                  | Return address of active display buffer   |
|                      |  | DISPLAY_BUFF1                   | Return address of display buffer number 1 |
|                      |  | DISPLAY_BUFF2                   | Return address of display buffer number 2 |
| <b>Return Value</b>  | ULONG  | Address of the specified buffer |   |
| <b>Description</b>   | <p>This function returns the address of one of the display buffers. The display driver manages two display buffers, Buff1 and Buff2. When double buffering is in use, one of these buffers is active and the other is inactive. A call to Display_ToggleBuffers() reverses the buffers. The active buffer is the one that the RAMDAC is receiving pixels from. An application should avoid writing to the active buffer because this may cause unwanted visual effects. Generally, an application calls Display_ToggleBuffers(), then immediately calls this function.</p> |                                 |   |
| <b>Example</b>       | <pre>ULONG InActiveBuff;<br/>Display_SetMode(640,480,60,DISPLAY_T555,DISPLAY_VIDEO);<br/>InActiveBuff = Display_GetBuffer(DISPLAY_INACTIVE);</pre>   |                                 |   |

|                      |   |
|----------------------|---|
| <b>Function Name</b> | <b>Display_GetMetrics</b>   |
| <b>Syntax</b>        | void Display_GetMetrics(DISPLAY_MET *M);  |
| <b>Arguments</b>     | <p>DISPLAY_MET *M Pointer to the DISPLAY_MET structure of metric parameters</p> <p>DISPLAY_MET consists of the following members:</p> <pre> BYTE   FtNumber /* 'C80 frame timer used: 0 or 1 */ BYTE   FclkRatio /* DCLK/FCLK ratio (default = 8) */ BYTE   RclkRatio /* DCLK/RCLK ratio */ BYTE   Bpp       /* Bits per pixel: 8, 16, or 32 */ USHORT Rh        /* Horizontal resolution (in pixels) */ USHORT Rv        /* Vertical resolution (in pixels) */ ULONG   Vram      /* Address of the start of VRAM */ ULONG   VramLen   /* Length of VRAM (in bytes) */ ULONG   Buff1     /* Frame buffer 1 address */ ULONG   Buff2     /* Frame buffer 2 address */ ULONG   Pitch     /* Frame buffer pitch (in bytes) */ ULONG   CurBuff   /* Current frame buffer address */ USHORT x         /* X-coord of display window (in pixels) */ USHORT y         /* Y-coord of display window (in pixels) */ USHORT dx        /* Width of display window (in pixels) */ USHORT dy        /* Height of display window (in pixels) */ float   Fv        /* Vertical frequency, refresh rate (in Hz) */ float   Fh        /* Horizontal frequency, line rate (in kHz) */ float   Fd        /* Pixel frequency, dot clock (in MHz) */ float   Ths       /* Horizontal sync interval (in μs) */ float   Thbp      /* Horizontal back porch interval (in μs) */ float   Tvs       /* Vertical sync interval (in μs) */ float   Tvbp      /* Vertical back porch interval (in μs) */ </pre> |
| <b>Return Value</b>  | None  |
| <b>Description</b>   | This function fills in the DISPLAY_MET structure pointed to by M with the current display state metrics.  |
| <b>Example</b>       | <pre> BOOL success; ULONG Pitch; DISPLAY_MET M;  Display_Init();  success = Display_SetMode(640, 480, 60, DISPLAY_P8,     DISPLAY_VIDEO);  if (success) {     Display_GetMetrics(&amp;M);     Pitch = M.Pitch;     Display_Enable();     /* ... do main programming ... */ }  Display_Disable(); </pre>   |

|                      |  |
|----------------------|--|
| <b>Function Name</b> | <b>Display_GetTvpRegs</b>  |
| <b>Syntax</b>        | void Display_GetTvpRegs(BYTE *R);  |
| <b>Arguments</b>     | BYTE *R      Pointer to a preallocated array of 64 bytes   |
| <b>Return Value</b>  | None   |
| <b>Description</b>   | This function reads the 64 internal RAMDAC registers and stores them into the array pointed to by R. The R argument must point to 64 bytes of preallocated memory. This function does not alter any of the registers. Refer to the <i>TVP3020 Video Interface Palette Data Manual</i> for register descriptions. |
| <b>Example</b>       | <pre>BYTE TvpRegs[64];  /* ... other setup ... */  Display_GetTvpRegs(R);  /* ... other processing ... */</pre>  |

|                      |  |
|----------------------|--|
| <b>Function Name</b> | <b>Display_Init</b>  |
| <b>Syntax</b>        | BOOL Display_Init();   |
| <b>Arguments</b>     | None   |
| <b>Return Value</b>  | BOOL            TRUE      Initialization succeeded<br>FALSE     Initialization failed  |
| <b>Description</b>   | <p>This function initializes the display hardware as follows:</p> <ul style="list-style-type: none"><li><input type="checkbox"/> Disables the 'C80 frame timer controller</li><li><input type="checkbox"/> Disables the RAMDAC</li><li><input type="checkbox"/> Writes all of the RAMDAC registers with zero</li><li><input type="checkbox"/> Programs the RAMDAC palette for grayscale</li><li><input type="checkbox"/> Sets the output mode to DISPLAY_VIDEO</li></ul> |
| <b>Example</b>       | <pre>Display_Init();<br/><br/>/* ... other processing ... */</pre>   |

|                      |   |
|----------------------|---|
| <b>Function Name</b> | <b>Display_InstallSema</b>  |
| <b>Syntax</b>        | long Display_InstallSema(long <i>Semald</i> );  |
| <b>Arguments</b>     | long <i>Semald</i> ID of an open semaphore as returned by TaskOpenSema()  |
| <b>Return Value</b>  | long              Previous semaphore ID   |
| <b>Description</b>   | <p>This function installs a semaphore into the display driver. The semaphore signals the application when a display buffer toggle request has completed. Toggling display buffers is only suitable during vertical blanking. Therefore, when the application calls Display_ToggleBuffers(), the toggle does not occur right away; rather, the toggle occurs at the next vertical blanking period. When the toggle occurs, the semaphore is signaled. This protocol allows the application to request a buffer toggle via Display_ToggleBuffers(), then wait until the toggle actually happens via TaskWaitSema(Semald).</p> <div><b>Note:</b><br/>The application must open the semaphore before calling this function. The semaphore is signaled only when the display is enabled.</div> |

**Example**

```
long DisplaySemaId;
ULONG Buff;

DisplaySemaId = TaskOpenSema(-1,0);

Display_Init();
Display_InstallSema(DisplaySemaId);
Display_SetMode(640,480,60,DISPLAY_T555,DISPLAY_VIDEO);
Display_Enable();

while (1) {
    Display_ToggleBuffers();
    TaskWaitSema(DisplaySemaId);
    Buff = Display_GetBuffer(DISPLAY_INACTIVE);

    /* do some processing here */
}
```

|                      |   |
|----------------------|---|
| <b>Function Name</b> | <b>Display_InstallTimingTable</b>   |
| <b>Syntax</b>        | void Display_InstallTimingTable(DISPLAY_MT *Table);   |
| <b>Arguments</b>     | DISPLAY_MT *Table     Pointer to a table of monitor timing structures.  |
| <b>Return Value</b>  | None  |
| <b>Description</b>   | This function installs a custom monitor timing table. If you wish to install a custom table into the driver, call this function before calling Display_SetMode(). Care must be taken when driving monitors with custom timing settings. You could copy the default table from the display driver module into an application and then modify it. |
| <b>Example</b>       | <pre>DISPLAY_MT MyTimingTable[] = {     { ... table data ... },     { ... table data ... },     { ... table data ... }, };  Display_Init(); <b>Display_InstallTimingTable(MyTimingTable);</b> Display_SetMode( 640, 480, 60, DISPLAY_T555, DISPLAY_VIDEO ); Display_Enable();</pre>   |



|                      |   |          |  |          |  |
|----------------------|---|----------|--|----------|--|
| <b>Function Name</b> | <b>Display_MoveWindow</b>   |          |  |          |  |
| <b>Syntax</b>        | <code>void Display_MoveWindow(USHORT x, USHORT y);</code>   |          |  |          |  |
| <b>Arguments</b>     | <table><tr><td>USHORT x</td><td>X-coordinate (top left corner) of display window (in pixels)</td></tr><tr><td>USHORT y</td><td>Y-coordinate (top left corner) of display window (in pixels)</td></tr></table>   | USHORT x | X-coordinate (top left corner) of display window (in pixels) | USHORT y | Y-coordinate (top left corner) of display window (in pixels) |
| USHORT x             | X-coordinate (top left corner) of display window (in pixels)  |          |  |          |  |
| USHORT y             | Y-coordinate (top left corner) of display window (in pixels)  |          |  |          |  |
| <b>Return Value</b>  | None  |          |  |          |  |
| <b>Description</b>   | <p>This function moves the display window to a new location on the screen.</p> <p>If the display hardware is enabled when you call this function, the function will wait until vertical blanking occurs before modifying any registers. No checking is done on the arguments, so your application must make sure the window is valid (that is, within full-screen resolution). Also, the horizontal pixel granularity is dependent on the FCLK (frame clock) ratio. The default is 4, so the horizontal window coordinate must be evenly divisible by 4.</p> <p>Before calling <code>Display_MoveWindow()</code>, you must first call <code>Display_SetWindow()</code>. See page 4-34 for more information about <code>Display_SetWindow()</code>. Both <code>Display_Init()</code> and <code>Display_SetMode()</code> reset the window parameters to their default state (<code>x = 0</code>, <code>y = 0</code>, <code>dx = Rh</code>, <code>dy = Rv</code>).</p> |          |  |          |  |
| <b>Example</b>       | <pre>BOOL success;  Display_Init();  success = Display_SetMode(1024, 768, 60, DISPLAY_TXRGB,     DISPLAY_VIDEO);  if (success) {     Display_SetWindow(0, 0, 512, 512);     <b>Display_MoveWindow(256, 128);</b>     Display_Enable();     /* ... other processing ... */ }  Display_Disable();</pre>   |          |  |          |  |

|                      |   |         |   |         |   |         |  |
|----------------------|---|---------|---|---------|---|---------|--|
| <b>Function Name</b> | <b>Display_ReadPalette</b>  |         |   |         |   |         |  |
| <b>Syntax</b>        | void Display_ReadPalette(BYTE *R, BYTE *G, BYTE *B);  |         |   |         |   |         |  |
| <b>Arguments</b>     | <table><tr><td>BYTE *R</td><td>Pointer to the variable that will receive a red value</td></tr><tr><td>BYTE *G</td><td>Pointer to the variable that will receive a green value</td></tr><tr><td>BYTE *B</td><td>Pointer to the variable that will receive a blue value</td></tr></table>   | BYTE *R | Pointer to the variable that will receive a red value | BYTE *G | Pointer to the variable that will receive a green value | BYTE *B | Pointer to the variable that will receive a blue value |
| BYTE *R              | Pointer to the variable that will receive a red value   |         |   |         |   |         |  |
| BYTE *G              | Pointer to the variable that will receive a green value   |         |   |         |   |         |  |
| BYTE *B              | Pointer to the variable that will receive a blue value  |         |   |         |   |         |  |
| <b>Return Value</b>  | None  |         |   |         |   |         |  |
| <b>Description</b>   | <p>This function reads an RGB triple (that is, three successive bytes of RGB data) from the current palette address in the RAMDAC. After each call, the palette address increments automatically. Normally, you should call Display_SetPaletteAddress() and then call Display_ReadPalette() repeatedly to read out the RAMDAC palette RAM. See page 4-29 for more information on Display_SetPaletteAddress(). You do not have to enable the display hardware in order to use Display_ReadPalette().</p> <div><b>Note:</b><br/>See page 4-33 for a listing of the RAMDAC palette RAM values for VGA colors (in RGB triples).</div> |         |   |         |   |         |  |
| <b>Example</b>       | <pre>BYTE R,G,B;  Display_SetPaletteAddress(0x00);  /* auto-address increment */ Display_ReadPalette(&amp;R, &amp;G, &amp;B); Display_ReadPalette(&amp;R, &amp;G, &amp;B); Display_ReadPalette(&amp;R, &amp;G, &amp;B);</pre>   |         |   |         |   |         |  |

|                      |  |
|----------------------|--|
| <b>Function Name</b> | <b>Display_SetBufferAddresses</b>  |
| <b>Syntax</b>        | void Display_SetBufferAddresses(ULONG <i>Buff1</i> , ULONG <i>Buff2</i> );   |
| <b>Arguments</b>     | <p>ULONG <i>Buff1</i>     VRAM address of frame buffer 1</p> <p>ULONG <i>Buff2</i>     VRAM address of frame buffer 2</p>  |
| <b>Return Value</b>  | None   |
| <b>Description</b>   | <p>Call this function if your application requires frame buffers to be located differently than the default. By default (by calling Display_SetMode()) the address of frame buffer 1 is set to the start of VRAM, and the address of frame buffer 2 is set to the midpoint in VRAM. Each time you call Display_ToggleBuffers(), the active display buffer toggles between these two buffers. See page 4-35 for more information on Display_ToggleBuffers().</p> <div style="border: 1px solid black; padding: 5px;"> <p><b>Note:</b> If you want to change the default frame buffer addresses, you must call Display_SetBufferAddresses():</p> <ul style="list-style-type: none"> <li><input type="checkbox"/> <i>after</i> calling Display_SetMode() (see page 4-25 for more information)</li> <li><input type="checkbox"/> <i>before</i> calling Display_Enable() (see page 4-12 for more information)</li> </ul> </div> |
| <b>Example</b>       | <pre> BOOL success;  success = Display_SetMode(800, 600, 60, DISPLAY_T565,     DISPLAY_VIDEO);  if (success) {     Display_SetBufferAddresses(0xC0000000, 0xC0100000);     Display_Enable();     /* ... other processing ... */ }  Display_Disable(); </pre>   |

|               |   |  |
|---------------|---|--|
| Function Name | Display_SetDotClock   |  |
| Syntax        | float Display_SetDotClock(float $F_d$ );  |  |
| Arguments     | float $F_d$   | The desired dot clock frequency:<br>10 MHz $\leq F_d \leq$ 170 MHz |
| Return Value  | float   | The actual dot clock frequency setting (in MHz)                    |
| Description   | <p>This function programs the pixel clock generator for the desired frequency (<math>F_d</math>). Because the clock generator operates at quantum levels, the actual frequency may differ slightly from the desired frequency. Generally, this difference is insignificant. The clock doubling input of the RAMDAC is used so the dot clock is actually programmed to half of that specified, then doubled. You should ignore this clock doubling and treat both the dot clock argument and return value as the true dot clock frequency.</p> <p>For example, if you require a dot clock of 64 MHz, you should call Display_SetDotClock(64.0). In this case, the function programs the pixel clock generator for 32.0 MHz so that the frequency is doubled in the RAMDAC to 64.0 MHz. The return value of this function would then be 64.0 MHz <math>\pm</math> quantization error.</p> <hr/> <p><b>Notes:</b></p> <ol style="list-style-type: none"><li>1) Generally, you should not call Display_SetDotClock() because it is called by Display_SetMode(). See page 4-25 for more information on Display_SetMode().</li><li>2) Call Display_SetDotClock() only for <i>custom</i> displays.</li></ol> |  |

**Example**

```
float ActualFrequency;

/* Program the dot clock to 63.7 MHz */
ActualFrequency = Display_SetDotClock(63.7);
```

**Function Name** **Display\_SetGreyScalePalette****Syntax** void DisplaySetGreyScalePalette();**Arguments** None**Return Value** None

**Description** This function programs the RAMDAC palette RAM for grayscale by setting each color cell in the palette array to its offset in the array. For instance, the cell at offset 0x7F is set to 0x7F. This is done for all three RAM arrays: red, green, and blue. The grayscale palette is the default.

**Notes:**

- 1) This function is called by Display\_Init() and Display\_SetMode().
- 2) The display does not have to be enabled to call this function, but you must call Display\_SetMode() first. See page 4-25 for more information on Display\_SetMode().

Refer to the following API functions for customizing the color palette:

| Function                    | See Page |
|-----------------------------|----------|
| Display_ReadPalette()       | 4-21     |
| Display_SetPaletteAddress() | 4-29     |
| Display_WritePalette()      | 4-39     |

**Example**

```
BOOL success;

success = Display_SetMode(800, 600, 60, DISPLAY_T565,
    DISPLAY_VIDEO);

if (success) {
    Display_SetGreyScalePalette();
    Display_Enable();
    /* ... other processing ... */
}

Display_Disable();
```

|                      |  |   |  |
|----------------------|--|---|--|
| <b>Function Name</b> | <b>Display_SetMode</b>   |   |  |
| <b>Syntax</b>        | BOOL Display_SetMode(USHORT <i>Rh</i> , USHORT <i>Rv</i> , float <i>Fv</i> , char * <i>PixFmt</i> , BYTE <i>Output</i> );  |   |  |
| <b>Arguments</b>     | USHORT <i>Rh</i>   | Horizontal resolution (in pixels)   |  |
|                      | USHORT <i>Rv</i>   | Vertical resolution (in pixels)   |  |
|                      | float <i>Fv</i>  | Vertical frequency refresh rate (in Hz)   |  |
|                      | char * <i>PixFmt</i>   | Pixel format designator (see Table 2–1 on page 2-8 for more detailed descriptions): |  |
|                      |  | DISPLAY_P8  | 8 BPP  |
|                      |  | DISPLAY_DXRGB32   | 32 BPP   |
|                      |  | DISPLAY_DBGRX   | 32 BPP   |
|                      |  | DISPLAY_D565  | 16 BPP   |
|                      |  | DISPLAY_D555  | 16 BPP   |
|                      |  | DISPLAY_D664  | 16 BPP   |
|                      |  | DISPLAY_D444  | 16 BPP   |
|                      |  | DISPLAY_TXRGB   | 32 BPP   |
|                      |  | DISPLAY_TBGRX32   | 32 BPP   |
|                      |  | DISPLAY_T565  | 16 BPP   |
|                      |  | DISPLAY_T555  | 16 BPP   |
|                      |  | DISPLAY_T664  | 16 BPP   |
|                      |  | DISPLAY_T444  | 16 BPP   |
|                      | BYTE <i>Output</i>   | Display output mode:  |  |
|                      |  | DISPLAY_PASSTHROUGH   | Select VGA pass-through                        |
|                      |  | DISPLAY_VIDEO   | Select RAMDAC output                           |
|                      |  | DISPLAY_OVERLAY   | Select overlaid video output                   |
| <b>Return Value</b>  | BOOL   | TRUE  | Mode is supported by monitor timing table.     |
|                      |  | FALSE   | Mode is not supported by monitor timing table. |
| <b>Description</b>   | <p>This function sets up a particular display mode. It is the most important function in the display API. For most applications, you need to use only the following functions to set up your display.</p> <ul style="list-style-type: none"> <li><input type="checkbox"/> Display_Init()</li> <li><input type="checkbox"/> Display_SetMode()</li> <li><input type="checkbox"/> Display_Enable()</li> <li><input type="checkbox"/> Display_Disable()</li> </ul> |   |  |

**Note:**

Because Display\_SetMode() disables the display hardware, you must call Display\_Enable() *after* Display\_SetMode() to reenable the display hardware.

Remember that a table of MT structures is used to obtain the monitor timing parameters. You supply  $R_h$ ,  $R_v$ , and  $F_v$ ; then the software uses these values to find a match in the table.

No argument checking is done to ensure that the parameters are valid. Therefore, your application must supply valid parameters.

**Unsupported Resolutions Could Damage Your Monitor**

Some monitors do not support the higher resolutions. Check your monitor specifications before attempting to drive it at a high resolution. Some monitors do not support resolutions greater than  $1024 \times 768$ . Remember also to verify the supported refresh rates.

If you have only 2M bytes of VRAM, you cannot set up a  $1024 \times 768$  display with 32 BPP (bits per pixel) because that would require  $1024 \times 768 \times 4 = 3\,145\,728$  bytes, which is more than 2M bytes. You can, however, set up window of  $512 \times 512$  at 32 BPP within the  $1024 \times 768$  display. This would require only  $512 \times 512 \times 4 = 1\,048\,576$  bytes, which is much less than 2M bytes.

**Example**

```
BOOL success;

success = Display_SetMode(640, 480, 60, DISPLAY_T565,
    DISPLAY_VIDEO);

if (success) {
    Display_Enable();
    /* ... other processing ... */
}

Display_Disable();
```

|                      |  |  |
|----------------------|--|--|
| <b>Function Name</b> | <b>Display_SetOverlayParams</b>  |  |
| <b>Syntax</b>        | void Display_SetOverlayParams(short <i>adx</i> , short <i>ady</i> , short <i>bdx</i> , short <i>bdy</i> , float <i>dThbp</i> , float <i>dTvbp</i> , float <i>Fd</i> , float <i>dFd</i> );  |  |
| <b>Arguments</b>     | short <i>adx</i>   | Signed offset to HSAREA (in pixels)      |
|                      | short <i>ady</i>   | Signed offset to VSAREA (in pixels)      |
|                      | short <i>bdx</i>   | Signed offset to HEBLNK (in pixels)      |
|                      | short <i>bdy</i>   | Signed offset to VEBLNK (in pixels)      |
|                      | float <i>dThbp</i>   | Signed offset to $t_{hbp}$ (in $\mu s$ ) |
|                      | float <i>dTvbp</i>   | Signed offset to $t_{vbp}$ (in $\mu s$ ) |
|                      | float <i>Fd</i>  | Replacement dot clock (in MHz)           |
|                      | float <i>dFd</i>   | Signed offset to the dot clock (in MHz)  |
| <b>Return Value</b>  | None   |  |
| <b>Description</b>   | <p>This function sets the video overlay parameters used when DISPLAY_OVERLAY mode is specified in a call to Display_SetMode(). The driver sets up the display like normal, then adjusts certain parameters to account for timing differences between the RAMDAC graphics and the VGA graphics input from the VGA pass-through cable. The AREA and BLNK signals are adjusted according to <i>adx</i>, <i>ady</i>, <i>bdx</i>, and <i>bdy</i>. The back porch timing parameters are adjusted according to <i>dThbp</i> and <i>dTvbp</i>. The <i>Fd</i> argument is special in that, if it is zero, the default dot clock is used. If <i>Fd</i> is nonzero, the <i>Fd</i> parameter is used instead of the default dot clock. Generally, the default dot clock is close to that of the VGA input from the PC graphics card, so <i>Fd</i> can be set to zero. The <i>dFd</i> parameter is a signed offset to the dot clock whether it is the default dot clock or the dot clock defined by the <i>Fd</i> parameter. A display window must be used when the display hardware is in overlay mode. The display resolution specified in Display_SetMode() should be the same as the VGA graphics card input.</p> |  |



**Example**

```
/* set up an overlay window onto a 640x480 display */
/* adx  = -16  pixels */
/* ady  = 0    pixels */
/* bdx  = 0    pixels */
/* bdy  = 12   pixels */
/* dThbp = 1.0   $\mu$ s */
/* dTvbp = -0.5  $\mu$ s */
/* Fd    = 0.0  MHz (use default dot clock) */
/* dFd    = 1.051 MHz */

Display_Init();
Display_SetOverlayParams(-16, 0, 0, 12, 1.0, -0.5, 0.0,
1.051);
Display_SetMode(640,480,60,DISPLAY_T555,DISPLAY_OVERLAY);
Display_SetWindow(96,96,128,64);
Display_Enable();
```

**Function Name**      **Display\_SetPaletteAddress****Syntax**                void Display\_SetPaletteAddress(BYTE *ad*);**Arguments**            BYTE *ad*                Offset into the RAMDAC palette RAM**Return Value**        None

**Description**                This function sets the read and write pointers of the color palette RAM to the specified address. Once these pointers are set, you can call one of the following functions:

| Function               | Description                      | See Page |
|------------------------|----------------------------------|----------|
| Display_ReadPalette()  | To read the color palette data   | 4-21     |
| Display_WritePalette() | To modify the color palette data | 4-39     |

**Example**

```
BYTE R,G,B;  
  
Display_SetPaletteAddress(0x00);  
  
Display_ReadPalette(&R, &G, &B);
```

|               |   |
|---------------|---|
| Function Name | <b>Display_SetPitch</b>   |
| Syntax        | <code>void Display_SetPitch(ULONG <i>pitch</i>);</code>   |
| Arguments     | ULONG <i>pitch</i> Desired frame buffer pitch (in bytes)  |
| Return Value  | None  |
| Description   | <p>This function sets the pitch of the display frame buffer. The <i>pitch</i> is the number of bytes from the start of one line to the start of the next line. By default, the pitch is set to the line width in bytes. For instance, if you call <code>Display_SetMode(1024, 768, 60, DISPLAY_T565, DISPLAY_VIDEO)</code>, then the pitch is <math>1024 \text{ pixels} \times 2 \text{ bytes/pixel} = 2048 \text{ bytes}</math>, or <code>0x0800</code> by default.</p> <p>Make sure that you do not try to set a pitch that is less than the line width (in bytes). Setting a pitch that is less than the line width may result in <i>semiomnipresent pixels</i> on the screen; that is, a particular pixel will appear to be at two locations on the screen at once. The number of lines in the active frame times the pitch is the amount of VRAM needed for a single frame buffer. For example, <math>1024 \times 2048 = 2\,097\,152 \text{ bytes} = 2\text{M bytes}</math>.</p> |
| Example       | <pre>BOOL success;  success = Display_SetMode(1024, 768, 60, DISPLAY_T565,     DISPLAY_VIDEO);  if (success) {     <b>Display_SetPitch(0x0800);</b>     Display_Enable();     /* ... other processing ... */ }  Display_Disable();</pre>  |

|                      |  |
|----------------------|--|
| <b>Function Name</b> | <b>Display_SetPixel</b>  |
| <b>Syntax</b>        | void Display_SetPixel(USHORT x, USHORT y, ULONG val, BYTE buffid);   |
| <b>Arguments</b>     | <p>USHORT x      x (horizontal) position of pixel to set</p> <p>USHORT y      y (vertical) position of pixel to set</p> <p>ULONG val     Value to store at pixel location</p> <p>BYTE buffid   Which display buffer to store value into:</p> <p>                DISPLAY_INACTIVE   Inactive display buffer</p> <p>                DISPLAY_ACTIVE     Active display buffer</p> <p>                DISPLAY_BUFF1      Display buffer number 1</p> <p>                DISPLAY_BUFF2      Display buffer number 2</p> |
| <b>Return Value</b>  | None   |
| <b>Description</b>   | This function is a utility used to set a single pixel in one of the display buffers. The function detects whether the display is set up for 8, 16, or 32 BPP and uses only that part of the val argument. This function is inefficient and should be used only for a test.   |
| <b>Example</b>       | <pre> USHORT x,y;  Display_SetMode(640,480,60,DISPLAY_T555,DISPLAY_VIDEO); Display_Enable();  /* fill entire active buffer with blue pixels one pixel at */ /* a time  */ for (x=0; x&lt;640; x++) {     for (y=0; y&lt;480; y++) {         Display_SetPixel(x, y ,0x001F, DISPLAY_ACTIVE);     } } </pre>   |

|                      |   |                |  |                |                                      |                 |                                       |
|----------------------|---|----------------|--|----------------|--------------------------------------|-----------------|---------------------------------------|
| <b>Function Name</b> | <b>Display_SetSyncPolarities</b>  |                |  |                |                                      |                 |                                       |
| <b>Syntax</b>        | <code>void Display_SetSyncPolarities(BYTE <i>Sh</i>, BYTE <i>Sv</i>, BYTE <i>Pen</i>);</code>   |                |  |                |                                      |                 |                                       |
| <b>Arguments</b>     | <table><tr><td>BYTE <i>Sh</i></td><td>Horizontal sync polarity: 0 = –, 1 = +</td></tr><tr><td>BYTE <i>Sv</i></td><td>Vertical sync polarity: 0 = –, 1 = +</td></tr><tr><td>BYTE <i>Pen</i></td><td>Pixel clock enable polarity: 0=–, 1=+</td></tr></table>  | BYTE <i>Sh</i> | Horizontal sync polarity: 0 = –, 1 = + | BYTE <i>Sv</i> | Vertical sync polarity: 0 = –, 1 = + | BYTE <i>Pen</i> | Pixel clock enable polarity: 0=–, 1=+ |
| BYTE <i>Sh</i>       | Horizontal sync polarity: 0 = –, 1 = +  |                |  |                |                                      |                 |                                       |
| BYTE <i>Sv</i>       | Vertical sync polarity: 0 = –, 1 = +  |                |  |                |                                      |                 |                                       |
| BYTE <i>Pen</i>      | Pixel clock enable polarity: 0=–, 1=+   |                |  |                |                                      |                 |                                       |
| <b>Return Value</b>  | None  |                |  |                |                                      |                 |                                       |
| <b>Description</b>   | <p>This function sets the polarities of the horizontal and vertical sync signals. By default, the sync polarities are set according to the parameters in the monitor timing (MT) table. (See Figure 4–1(a) on page 4-4 for the definition of the MT data structure.) Set an argument to 0 to invert the sync signal (negative sync) or set it to 1 for positive sync. The Pen argument is used to invert the pixel clock generator enable signal.</p> |                |  |                |                                      |                 |                                       |
| <b>Example</b>       | <pre>BOOL success;  success = Display_SetMode(640, 480, 60, DISPLAY_T565,     DISPLAY_VIDEO);  if (success) {     <b>Display_SetSyncPolarities(0,1,0);</b>     Display_Enable();     /* ... other processing ... */ }  Display_Disable();</pre>   |                |  |                |                                      |                 |                                       |

**Function Name** **Display\_SetVgaPalette****Syntax** void Display\_SetVgaPalette();**Arguments** None**Return Value** None

**Description** This function programs the RAMDAC palette RAM for VGA colors. Only 16 VGA colors are defined, so they are repeated 16 times to fill the color palette RAM ( $16 \times 16 = 256$ ). The display does not have to be enabled before you call this function, but you must call Display\_SetMode() first. See page 4-25 for more information on Display\_SetMode().

The following table lists the RAMDAC's palette RAM values for VGA colors (in RGB triples):

|    | Color         | Red  | Green | Blue |
|----|---------------|------|-------|------|
| 0  | Black         | 0x00 | 0x00  | 0x00 |
| 1  | Blue          | 0x00 | 0x00  | 0x80 |
| 2  | Green         | 0x00 | 0x80  | 0x00 |
| 3  | Cyan          | 0x00 | 0x80  | 0x80 |
| 4  | Red           | 0x80 | 0x00  | 0x00 |
| 5  | Magenta       | 0x80 | 0x00  | 0x80 |
| 6  | Brown         | 0x80 | 0x80  | 0x00 |
| 7  | Grey          | 0x40 | 0x40  | 0x40 |
| 8  | Light Grey    | 0x80 | 0x80  | 0x80 |
| 9  | Light Blue    | 0x00 | 0x00  | 0xFF |
| 10 | Light Green   | 0x00 | 0xFF  | 0x00 |
| 11 | Light Cyan    | 0x00 | 0xFF  | 0xFF |
| 12 | Light Red     | 0xFF | 0x00  | 0x00 |
| 13 | Light Magenta | 0xFF | 0x00  | 0xFF |
| 14 | Yellow        | 0xFF | 0xFF  | 0x00 |
| 15 | White         | 0xFF | 0xFF  | 0xFF |

**Example**

```

BOOL success;

success = Display_SetMode(800, 600, 60, DISPLAY_P8,
    DISPLAY_VIDEO);

if (success) {
    Display_SetVgaPalette();
    Display_Enable();
    /* ... other processing ... */
}
Display_Disable();

```

|                      |  |
|----------------------|--|
| <b>Function Name</b> | <b>Display_SetWindow</b>   |
| <b>Syntax</b>        | void Display_SetWindow(USHORT x, USHORT y, USHORT dx, USHORT dy);  |
| <b>Arguments</b>     | <p>USHORT x    X-coordinate (top left corner) of the display window (in pixels)</p> <p>USHORT y    Y-coordinate (top left corner) of the display window (in pixels)</p> <p>USHORT dx   Width of window (in pixels)</p> <p>USHORT dy   Height of window (in pixels)</p>   |
| <b>Return Value</b>  | None   |
| <b>Description</b>   | <p>This function sets the display window. If the display hardware is enabled when you call this function, the function waits until vertical blanking occurs before modifying any registers. No checking is done on the arguments, so you must make sure the window is valid. Both Display_Init() and Display_SetMode() reset the window parameters to their default state (x = 0, y = 0, dx = Rh, dy = Rv).</p> <hr/> <p><b>Note:</b></p> <p>The horizontal pixel granularity is dependent on the FCLK (frame clock) ratio. The default is 4, so the horizontal window coordinate must be evenly divisible by 4.</p> <hr/> |
| <b>Example</b>       | <pre> BOOL success;  Display_Init();  success = Display_SetMode(1024, 768, 60, DISPLAY_TXRGB,     DISPLAY_VIDEO);  if (success) {     Display_SetWindow(0, 0, 512, 512);     Display_MoveWindow(256, 128);     /* ... other processing ... */ }  Display_Disable(); </pre>   |

|                      |  |
|----------------------|--|
| <b>Function Name</b> | <b>Display_ToggleBuffers</b>   |
| <b>Syntax</b>        | void Display_ToggleBuffers();  |
| <b>Arguments</b>     | None   |
| <b>Return Value</b>  | None   |
| <b>Description</b>   | <p>This function tells the display driver to toggle the active display buffer at the next vertical blanking period. When the toggle takes place, the driver signals the display semaphore. The display driver manages two display buffers, Buff1 and Buff2. When double buffering is in use, one of these buffers is active and the other is inactive. A call to Display_ToggleBuffers() reverses the buffers. The active buffer is the one that the RAMDAC is receiving pixels from. An application should avoid writing to the active buffer because this may cause unwanted visual effects. Generally, an application calls Display_ToggleBuffers(), then waits on the display semaphore before calling Display_GetBuffer() to get the new inactive buffer.</p> |
| <b>Example</b>       | <pre>long DisplaySemaId; ULONG Buff;  DisplaySemaId = TaskOpenSema(-1,0); Display_Init(); Display_InstallSema(DisplaySemaId); Display_SetMode(640,480,60,DISPLAY_T555,DISPLAY_VIDEO); Display_Enable();  while (1) {     <b>Display_ToggleBuffers();</b>     TaskWaitSema(DisplaySemaId);     Buff = Display_GetBuffer(DISPLAY_INACTIVE);      /* do some processing here */ }</pre>   |



|                      |  |
|----------------------|--|
| <b>Function Name</b> | <b>Display_TvpRegIn</b>  |
| <b>Syntax</b>        | BYTE Display_TvpRegIn(BYTE <i>reg</i> );   |
| <b>Arguments</b>     | BYTE <i>reg</i> Palette internal register number   |
| <b>Return Value</b>  | BYTE            Value of register  |
| <b>Description</b>   | This function reads a value from one of the RAMDAC internal registers. Refer to the <i>TVP3020 Video Interface Palette Data Manual</i> for a listing of the registers. |
| <b>Example</b>       | <pre>BYTE val;<br/><br/>val = Display_TvpRegIn(0x1B);<br/>Display_TvpRegOut(0x1B, val   0x80);</pre>   |

|                      |   |
|----------------------|---|
| <b>Function Name</b> | <b>Display_TvpRegOut</b>  |
| <b>Syntax</b>        | void Display_TvpRegOut(BYTE <i>reg</i> , BYTE <i>val</i> );   |
| <b>Arguments</b>     | BYTE <i>reg</i> Palette internal register number<br>BYTE <i>val</i> Value of register   |
| <b>Return Value</b>  | None  |
| <b>Description</b>   | This function writes a value to one of the RAMDAC internal registers. Refer to the <i>TVP3020 Video Interface Palette Data Manual</i> for a listing of the registers. |
| <b>Example</b>       | <pre>BYTE val;<br/><br/>val = Display_TvpRegIn(0x1B);<br/>Display_TvpRegOut(0x1B, val   0x80);</pre>  |

|                      |   |
|----------------------|---|
| <b>Function Name</b> | <b>Display_WaitEndOfFrame</b>   |
| <b>Syntax</b>        | <code>void Display_WaitEndOfFrame();</code>   |
| <b>Arguments</b>     | None  |
| <b>Return Value</b>  | None  |
| <b>Description</b>   | This function waits until vertical blanking occurs and then returns. <code>Display_WaitEndOfFrame()</code> first clears the frame timer interrupt pending bit and then polls it until the bit is set. If the display hardware is disabled, this function returns without waiting.             |
| <b>Example</b>       | <pre>BOOL success;  Display_Init();  success = Display_SetMode(1024, 768, 60, DISPLAY_TXRGB,     DISPLAY_VIDEO);  if (success) {     Display_Enable();     /* ... other processing ... */     <b>Display_WaitEndOfFrame();</b>     /* ... other processing ... */ }  Display_Disable();</pre> |

|                      |   |               |           |               |             |               |            |
|----------------------|---|---------------|-----------|---------------|-------------|---------------|------------|
| <b>Function Name</b> | <b>Display_WritePalette</b>   |               |           |               |             |               |            |
| <b>Syntax</b>        | <code>void Display_WritePalette(BYTE R, BYTE G, BYTE B);</code>   |               |           |               |             |               |            |
| <b>Arguments</b>     | <table><tr><td>BYTE <i>R</i></td><td>Red value</td></tr><tr><td>BYTE <i>G</i></td><td>Green value</td></tr><tr><td>BYTE <i>B</i></td><td>Blue value</td></tr></table>   | BYTE <i>R</i> | Red value | BYTE <i>G</i> | Green value | BYTE <i>B</i> | Blue value |
| BYTE <i>R</i>        | Red value   |               |           |               |             |               |            |
| BYTE <i>G</i>        | Green value   |               |           |               |             |               |            |
| BYTE <i>B</i>        | Blue value  |               |           |               |             |               |            |
| <b>Return Value</b>  | None  |               |           |               |             |               |            |
| <b>Description</b>   | <p>This function writes an RGB triple to the current palette address in the RAMDAC. After each call, the palette address automatically increments. Normally, you would call <code>Display_SetPaletteAddress()</code> and then call <code>Display_WritePalette()</code> repeatedly to fill the palette RAM. See page 4-29 for more information on <code>Display_SetPaletteAddress()</code>.</p> <div><b>Note:</b><br/>The display hardware does not have to be enabled before you use this function.</div> |               |           |               |             |               |            |
| <b>Example</b>       | <pre>Display_SetPaletteAddress(0x50);<br/>Display_WritePalette(0x20, 0x10, 0x80);<br/>Display_WritePalette(0x40, 0x20, 0x40);</pre>   |               |           |               |             |               |            |

# Video Capture API

---

---

---

This chapter discusses the video capture macros and data type. It also describes, in alphabetical order, the application programming interface (API) functions associated with the video capture driver for the TMS320C8x software development board (SDB).

| <b>Topic</b>   | <b>Page</b> |
|--|-------------|
| <b>5.1 Video Capture API Macros and Data Types .....</b> | <b>5-2</b>  |
| <b>5.2 Video Capture Buffering .....</b>                 | <b>5-5</b>  |
| <b>5.3 Video Capture API Functions .....</b>             | <b>5-8</b>  |

## 5.1 Video Capture API Macros and Data Types

Table 5–1 describes the macros used by the video capture API and lists the API functions that use each macro. Figure 5–1 provides the definition for the video capture API data type. These macros and the data type, as well the API function prototypes, are defined in <capture.h>. The object code resides in sdbdrv.lib.

*Table 5–1. Video Capture API Macros*

*(a) Input source constants*

| Macro Name           | Value | Description         |
|----------------------|-------|---------------------|
| #define CAPTURE_SVHS | 0x00  | S-VHS input source  |
| #define CAPTURE_VID1 | 0x01  | CVBS input source 1 |
| #define CAPTURE_VID2 | 0x03  | CVBS input source 2 |

**Note:** These macros are used by the function `Capture_SetInputSource()`.

*(b) Input format constants*

| Macro Name           | Value | Description         |
|----------------------|-------|---------------------|
| #define CAPTURE_NTSC | 0x01  | Select NTSC setting |
| #define CAPTURE_PAL  | 0x02  | Select PAL setting  |

**Note:** These macros are used by the function `Capture_Install()`.

*(c) Scaling constants*

| Macro Name              | Value | Description                |
|-------------------------|-------|----------------------------|
| #define CAPTURE_640x480 | 0x01  | 640 × 480 interlaced       |
| #define CAPTURE_512x512 | 0x02  | 512 × 512 interlaced       |
| #define CAPTURE_CIF     | 0x03  | 352 × 288 interlaced       |
| #define CAPTURE_CIFK    | 0x04  | 352 × 240 even fields only |
| #define CAPTURE_QCIF    | 0x05  | 176 × 144 even fields only |
| #define CAPTURE_SQCIF   | 0x06  | 128 × 96 even fields only  |

**Note:** These macros are used by the function `Capture_Install()`.

Table 5–1. Video Capture API Macros (Continued)

(d) Output format constants

| Macro Name             | Value | Description   |
|------------------------|-------|---|
| #define CAPTURE_YUV422 | 0x01  | Multiplexed YUV 4:2:2 format: YUYVYUYV ...                |
| #define CAPTURE_RGB888 | 0x02  | RGB 8,8,8,X format: RGBXRGBX ...                          |
| #define CAPTURE_RGB555 | 0x03  | RGB $\alpha$ ,5,5,5 format: $\alpha$ RGB $\alpha$ RGB ... |
| #define CAPTURE_MONO8  | 0x04  | 8-bit monochrome (luminance only) format: YYYYYYYY ...    |

**Note:** These macros are used by the function Capture\_Install().

Figure 5–1. Video Capture API Data Type (Metrics Parameter Structure CAPTURE\_MET)

```
typedef struct {
    BYTE  Fps;          /* frames per second */
    ULONG Size;         /* buffer size in bytes */
    USHORT Rh;         /* horizontal capture resolution in pixels */
    USHORT Rv;         /* vertical resolution in lines */
    BYTE  Bpp;         /* bits per pixel */
    ULONG Pitch;       /* buffer pitch, number of bytes from one line to next */
    BOOL  Interlace;   /* interlaced flag (TRUE = interlaced) */
} CAPTURE_MET;
```

### 5.1.1 Supported Scaling Resolutions

The six API-supported video capture scaling resolutions are listed in Table 5–1(c). Two of these resolutions have special limitations: CAPTURE\_512x512 (which has a pixel resolution of  $512 \times 512$ ) and CAPTURE\_CIFK (which has a pixel resolution of  $352 \times 288$ ). The video capture hardware captures video at a resolution of  $640 \times 480$  pixels in interlaced mode. The hardware scaler does not have the ability to upscale the image; it can only downscale. Because of this upscaling limitation, 512 lines of video, as needed by CAPTURE\_512x512, cannot be created from 480 lines. However, because  $512 \times 512$  is a common video resolution, the API simulates it by capturing  $512 \times 480$  into a  $512 \times 512$  buffer. Thus, the application gets a  $512 \times 512$  buffer, but only the first 480 lines contain video—the other 32 lines are undefined. You can clear out the entire buffer via `Capture_FillBufs()`, then the 32 lines at the bottom of the buffer will remain cleared unless the application modifies them.

The  $352 \times 288$  resolution faces the same limitation for noninterlaced mode, which uses only half of the captured image (a maximum of  $480 \text{ lines} / 2$  or 240 lines). The same method used to overcome the  $512 \times 512$  limitation is also applied to the  $352 \times 288$  resolution. The API creates buffers that are  $352 \times 288$  but only captures  $352 \times 240$  noninterlaced video into the buffers.

### 5.1.2 Video Capture Metric Parameters

To get information about the current video capture subsystem, call `Capture_GetMetrics()` to get a pointer to a `CAPTURE_MET` structure. Figure 5–1 gives the type definition for `CAPTURE_MET`. Refer to page 5-16 for more information on `Capture_GetMetrics()`.



## 5.2 Video Capture Buffering

The video capture driver manages a double buffering scheme internal to the driver. When the application calls `Capture_Install()`, the driver creates an internal buffering structure that contains two storage buffers: `Buff1` and `Buff2`. These two buffers are dynamically allocated on the heap and are used to store the captured video as it comes in one line at a time. At any given time, one of these buffers is the active buffer and the other one is the inactive buffer. The active buffer is the one currently receiving new lines of video. The application cannot access the active buffer.

The video capture hardware generates an interrupt every time a new frame is detected by an odd-to-even field transition of the video input. The interrupt service routine (ISR) toggles the active and inactive buffers; that is, the active buffer becomes inactive and the inactive buffer becomes active. This behavior is effectively a ping-pong action in which the buffers are toggled each frame. As a result, the active buffer always receives incoming video, and the inactive buffer has the most recent frame of video. This behavior continues, free-running, until the application interferes by requesting a buffer.

When the application needs a frame of newly captured video, it calls `Capture_GetBuffer()`, which returns the address of the inactive buffer. This interferes with the ISR ping-ponging because the application cannot use a buffer at the same time that the ISR uses it. For this reason, the driver manages the `InUse` flag, which is set when the application calls `Capture_GetBuffer()`. The ISR does not toggle the buffers when the `InUse` flag is set. The result is the active buffer does not change and all new frames of incoming video get stored into the same buffer (the active one). Ping-ponging then stops. The application has effectively locked the inactive buffer. The locking of the inactive buffer allows the application to modify the contents of the buffer as necessary. When the application is finished with the buffer, the buffer operation needs to be put back to normal. The application must return the buffer acquired from `GetCaptureBuffer()` to the ISR by calling `Capture_FreeBuffer()`. The `Capture_FreeBuffer()` function clears the `InUse` flag and ISR ping-ponging resumes.

The buffering mechanism just described has a hazard that occurs when the application performs the following sequence of events:

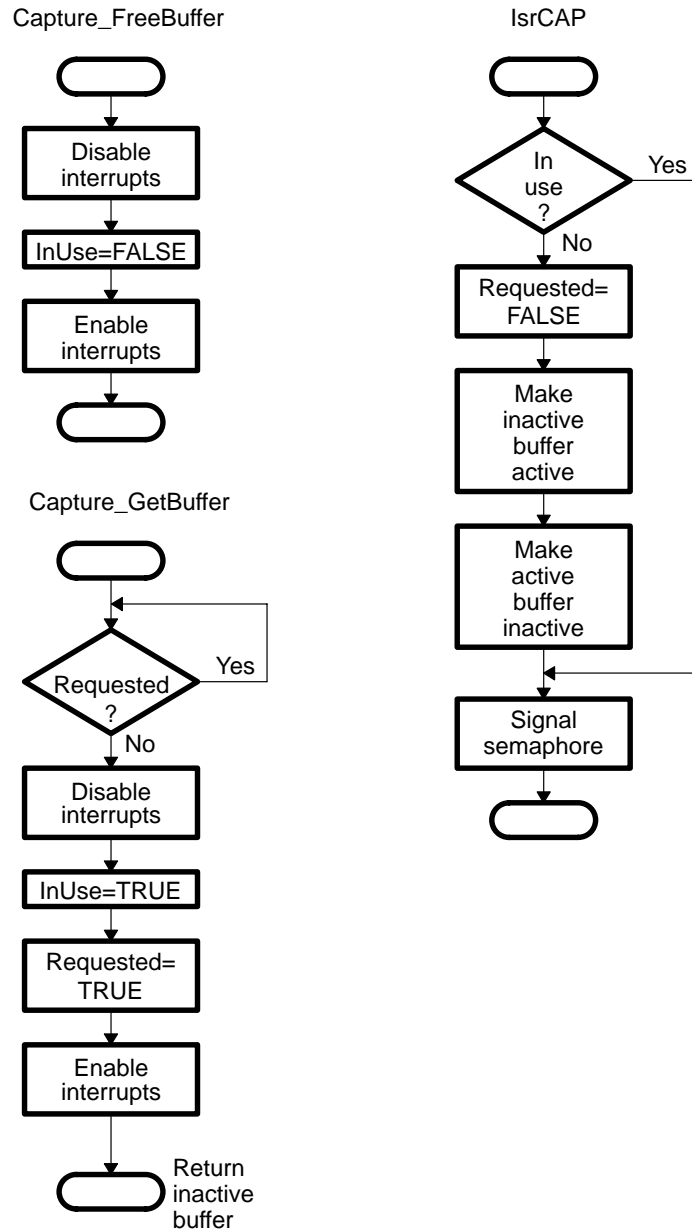
- 1) Calls `Capture_GetBuffer()`, which returns the inactive buffer
- 2) Modifies the contents of the inactive buffer
- 3) Calls `Capture_FreeBuffer()`, which returns the buffer to the ISR
- 4) Calls `Capture_GetBuffer()` again before a new frame has arrived since the call to `Capture_FreeBuffer()`

After this sequence of events, the buffer obtained by calling `Capture_GetBuffer()` in step 4 contains corrupted data that was modified in step 2. This happens because the `Capture_GetBuffer()` function was called in step 4 before an actual new frame of video was captured by the hardware. Because modifying the buffer contents is a common practice (color space conversion, for example), a mechanism has been built into the API to avoid corrupted buffers being passed to the application. The built-in fix causes the `Capture_GetBuffer()` function, as called in step 4, to stall until a new frame arrives.

The driver manages a second flag, `Requested`. Whenever the ISR toggles the buffers, it clears the `Requested` flag. The `Capture_GetBuffer()` function spins until the `Requested` flag is cleared by the ISR. Then, the function sets the `Requested` flag. As a result, a call to `Capture_GetBuffer()` will wait until a new buffer is ready before obtaining the inactive buffer. The only time this wait occurs is when the application is trying to obtain buffers faster than they are available.

Figure 5–2 better illustrates the double buffering using flow charts. These flow charts show the general flow and program logic; the actual variable names and identifiers may differ in the driver code. Also, nonrelated parts of the code are not shown.

Figure 5–2. Video Capture Double Buffering Logic



## 5.3 Video Capture API Functions

Listed below in alphabetical order are the video capture API functions. Use this list as a table of contents to the video capture API functions.

| Function                     | Page |
|------------------------------|------|
| Capture_CardPresent .....    | 5-9  |
| Capture_Disable .....        | 5-10 |
| Capture_Enable .....         | 5-11 |
| Capture_FillBufs .....       | 5-12 |
| Capture_FreeBuffer .....     | 5-13 |
| Capture_GetBuffer .....      | 5-14 |
| Capture_GetDecoderRegs ..... | 5-15 |
| Capture_GetMetrics .....     | 5-16 |
| Capture_GetScalerRegs .....  | 5-17 |
| Capture_Init .....           | 5-18 |
| Capture_Install .....        | 5-19 |
| Capture_InstallSema .....    | 5-21 |
| Capture_SetInputSource ..... | 5-22 |
| Capture_SetScaling .....     | 5-23 |
| Capture_UnInstall .....      | 5-24 |

|                      |  |
|----------------------|--|
| <b>Function Name</b> | <b>Capture_CardPresent</b>   |
| <b>Syntax</b>        | BOOL Capture_CardPresent();  |
| <b>Arguments</b>     | None   |
| <b>Return Value</b>  | BOOL            TRUE     Video capture card detected<br>FALSE    Video capture card not detected   |
| <b>Description</b>   | This function reads the video capture card ID register (CAPID) and returns TRUE if it is present; otherwise, it returns FALSE.   |
| <b>Example</b>       | <pre>Capture_Init();  if (Capture_CardPresent()) {     Capture_Install(arguments);     Capture_Enable();     /*...do main programming...*/     Capture_Disable();     Capture_UnInstall(); }</pre> |

|                      |  |
|----------------------|--|
| <b>Function Name</b> | <b>Capture_Disable</b>   |
| <b>Syntax</b>        | void Capture_Disable();  |
| <b>Arguments</b>     | None   |
| <b>Return Value</b>  | None   |
| <b>Description</b>   | <p>This function disables video capture by disabling frame and row hardware events.</p> <div><b>Note:</b><br/>Before calling Capture_Disable(), you must first call Capture_Install(). See page 5-19 for more information on Capture_Install().</div>  |
| <b>Example</b>       | <pre>BOOL B;<br/><br/>Capture_Init();<br/><br/>if (Capture_CardPresent()) {<br/>    B = Capture_Install(arguments);<br/><br/>    if (B) {<br/>        Capture_Enable();<br/>        /* ... do main programming ... */<br/>        <b>Capture_Disable();</b><br/>        Capture_UnInstall();<br/>    }<br/>}</pre> |

|                      |  |
|----------------------|--|
| <b>Function Name</b> | <b>Capture_Enable</b>  |
| <b>Syntax</b>        | void Capture_Enable();   |
| <b>Arguments</b>     | None   |
| <b>Return Value</b>  | None   |
| <b>Description</b>   | This function enables video capture by enabling frame and row hardware events.<br><br><div><b>Note:</b><br/>Before calling Capture_Enable(), you must first call Capture_Install(). See page 5-19 for more information on Capture_Install().</div> |

**Example**

```
BOOL B;  
  
Capture_Init();  
  
if (Capture_CardPresent()) {  
    B = Capture_Install(arguments);  
  
    if (B) {  
        Capture_Enable();  
        /* ... do main programming ... */  
        Capture_Disable();  
        Capture_UnInstall();  
    }  
}
```

|                      |  |
|----------------------|--|
| <b>Function Name</b> | <b>Capture_FillBufs</b>  |
| <b>Syntax</b>        | <code>void Capture_FillBufs(ULONG val);</code>   |
| <b>Arguments</b>     | ULONG <i>val</i> 32-bit value used to fill the buffers   |
| <b>Return Value</b>  | None   |
| <b>Description</b>   | <p>This function fills the two internal capture buffers (active and inactive) with the value specified.</p> <div><b>Note:</b><br/>Do not call Capture_FillBufs() while video capture is enabled.</div> |
| <b>Example</b>       | <pre>Capture_Install(CAPTURE_NTSC, CAPTURE_YUV422, CAPTURE_QCIF);<br/><b>Capture_FillBufs(0x00000000);</b><br/>Capture_Enable();</pre>   |



|                      |   |
|----------------------|---|
| <b>Function Name</b> | <b>Capture_FreeBuffer</b>   |
| <b>Syntax</b>        | void Capture_FreeBuffer();  |
| <b>Arguments</b>     | None  |
| <b>Return Value</b>  | None  |
| <b>Description</b>   | <p>This function frees the buffer obtained by the last call to Capture_GetBuffer(). Freeing the buffer allows the video capture ISR to start ping-ponging between two buffers again. Once an application has called Capture_GetBuffer(), it cannot call it again until it has called Capture_FreeBuffer(). For a more detailed explanation of the video capture buffers, see Section 5.2.</p> |
| <b>Example</b>       | <pre>long  CaptureSemaId; ULONG Buff;  Capture_Init(); CaptureSemaId = TaskOpenSema(-1,0); Capture_InstallSema(CaptureSemaId); Capture_Enable();  while (1) {     TaskWaitSema(CaptureSemaId);     Buff = Capture_GetBuffer();      /* do processing here */      Capture_FreeBuffer(); }</pre>   |

|                      |   |
|----------------------|---|
| <b>Function Name</b> | <b>Capture_GetBuffer</b>  |
| <b>Syntax</b>        | ULONG Capture_GetBuffer();  |
| <b>Arguments</b>     | None  |
| <b>Return Value</b>  | ULONG            Address of inactive buffer   |
| <b>Description</b>   | <p>This function returns the address of the inactive video capture buffer and locks the buffer for exclusive use by the application. The application can modify the contents of the buffer. When the application is finished using the buffer, it must call Capture_FreeBuffer() to return the buffer to the driver. For a more detailed explanation of the video capture buffers, see Section 5.2.</p> |
| <b>Example</b>       | <pre>long  CaptureSemaId; ULONG Buff;  Capture_Init(); Capture_Install(CAPTURE_NTSC,CAPTURE_RGB888,CAPTURE_640x480); CaptureSemaId = TaskOpenSema(-1,0); Capture_InstallSema(CaptureSemaId); Capture_Enable();  while (1) {     TaskWaitSema(CaptureSemaId);     <b>Buff = Capture_GetBuffer();</b>      /* do processing here */      Capture_FreeBuffer(); }</pre>                                    |

|                      |  |
|----------------------|--|
| <b>Function Name</b> | <b>Capture_GetDecoderRegs</b>  |
| <b>Syntax</b>        | void Capture_GetDecoderRegs(BYTE *DR);   |
| <b>Arguments</b>     | BYTE *DR      Pointer to a preallocated array of 25 bytes that will be filled in with video decoder register values  |
| <b>Return Value</b>  | None   |
| <b>Description</b>   | This function fills in the array of bytes pointed to by DR with the current video decoder register settings. Because there are a total of 25 decoder registers, DR must point to 25 bytes of preallocated memory. Although the decoder registers cannot be read directly, the hardware layer of the software keeps track of register writes and maintains a registry of the current register values. |
|                      | <hr/> <b>Notes:</b> <ol style="list-style-type: none"> <li>1) Use this function for reference and debugging purposes.</li> <li>2) Before calling Capture_GetDecoderRegs(), you must first call Capture_Init(). See page 5-18 for more information on Capture_Init().</li> </ol> <hr/>  |
| <b>Example</b>       | <pre> BYTE Dregs[25]; BYTE *buff; BOOL B;  Capture_Init();  if (Capture_CardPresent()) {     B = Capture_Install(arguments);     if (B) {         Capture_GetDecoderRegs(Dregs);         Capture_Enable();         /* ... do main programming ... */         Capture_Disable();         Capture_UnInstall();     } } </pre>  |

|                      |   |
|----------------------|---|
| <b>Function Name</b> | <b>Capture_GetMetrics</b>   |
| <b>Syntax</b>        | void Capture_GetMetrics(CAPTURE_MET *M);  |
| <b>Arguments</b>     | <p>CAPTURE_MET *M Pointer to the CAPTURE_MET structure of video capture metric parameters</p> <p>CAPTURE_MET consists of the following members :</p> <pre> BYTE   Fps;           /* frames per second                */ ULONG  Size;          /* buffer size in bytes             */ USHORT Rh;            /* horizontal capture resolution in pixels*/ USHORT Rv;            /* vertical resolution in lines      */ BYTE   Bpp;           /* bits per pixel                   */ ULONG  Pitch;         /* buffer pitch, number of bytes from one                         line to next */ BOOL   Interlace;     /* interlaced flag (TRUE = interlaced) */ </pre> |
| <b>Return Value</b>  | None  |
| <b>Description</b>   | <p>This function fills in the CAPTURE_MET structure that is pointed to by M with the current capture state metrics.</p> <div style="border: 1px solid black; padding: 5px; margin-top: 10px;"> <p><b>Note:</b></p> <p>Before calling Capture_GetMetrics(), you must first call Capture_Install(). See page 5-19 for more information on Capture_Install().</p> </div>   |
| <b>Example</b>       | <pre> CAPTURE_MET cmet; BOOL B;  Capture_Init();  if (Capture_CardPresent()) {     B = Capture_Install(arguments);     if (B) {         Capture_GetMetrics(&amp;cmet);         Capture_Enable();         /* ... do main programming ... */         Capture_Disable();         Capture_UnInstall();     } } </pre>   |

|                      |   |
|----------------------|---|
| <b>Function Name</b> | <b>Capture_GetScalerRegs</b>  |
| <b>Syntax</b>        | void Capture_GetScalerRegs(BYTE *SR);   |
| <b>Arguments</b>     | BYTE *SR      Pointer to a preallocated array of 17 bytes (unsigned characters) that will be filled in with scaler register values  |
| <b>Return Value</b>  | None  |
| <b>Description</b>   | <p>This function fills in the array of bytes that is pointed to by SR with the current video scaler register settings. Because there are a total of 17 scaler registers, SR must point to 17 bytes of preallocated memory. Although the scaler registers cannot be read directly, the hardware layer of the software keeps track of register writes and maintains a registry of the current register values.</p> <hr/> <p><b>Notes:</b></p> <ol style="list-style-type: none"><li>1) Use this function for reference and debugging purposes.</li><li>2) Before calling Capture_GetScalerRegs(), you must first call Capture_Init(). See page 5-18 for more information on Capture_Init().</li></ol> <hr/> |
| <b>Example</b>       | <pre>BYTE Sregs[25]; BYTE *buff; BOOL B;  Capture_Init();  if (Capture_CardPresent()) {     B = Capture_Install(arguments);     if (B) {         Capture_GetScalerRegs(Sregs);         Capture_Enable();         /* ... do main programming ... */         Capture_Disable();         Capture_UnInstall();     } }</pre>  |

|               |  |               |   |
|---------------|--|---------------|---|
| Function Name | Capture_Init   |               |   |
| Syntax        | BOOL Capture_Init();   |               |   |
| Arguments     | None   |               |   |
| Return Value  | BOOL   | TRUE<br>FALSE | Initialization succeeded<br>Initialization failed |
| Description   | <p>This function initializes the video capture hardware to a known state. Generally, this function should be called only once—at the beginning of a program.</p> <div><b>Note:</b><br/>You must call Capture_Init() <i>before</i> calling any other video capture function listed in this API.</div> |               |   |
| Example       | <pre>BOOL B;<br/><br/>Capture_Init();<br/><br/>if (Capture_CardPresent()) {<br/>    B = Capture_Install(arguments);<br/>    Capture_Enable();<br/>    /* ... do main programming ... */<br/>    Capture_Disable();<br/>    Capture_UnInstall();<br/>}</pre>  |               |   |

|   |   |                            |                                   |
|---|---|----------------------------|-----------------------------------|
| Function Name   | Capture_Install   |                            |                                   |
| Syntax  | BOOL Capture_Install(BYTE <i>InFormat</i> , BYTE <i>OutFormat</i> , BYTE <i>Scaling</i> );  |                            |                                   |
| Arguments   | BYTE <i>InFormat</i>  | Video input format:        |                                   |
|   |   | CAPTURE_NTSC               | Select NTSC setting               |
|   | CAPTURE_PAL   | Select PAL setting         |                                   |
|   | BYTE <i>OutFormat</i>   | Video output format:       |                                   |
|   |   | CAPTURE_YUV422             | Multiplexed YUV 4:2:2 format      |
|   |   | CAPTURE_RGB888             | RGB 8,8,8,X format                |
|   |   | CAPTURE_RGB555             | RGB α,5,5,5 format                |
|   |   | CAPTURE_MONO8              | 8-bit monochrome (luminance only) |
|   | BYTE <i>Scaling</i>   | Scaling constant:          |                                   |
|   |   | CAPTURE_640x480            | 640 × 480 interlaced              |
|   |   | CAPTURE_512x512            | 512 × 512 interlaced              |
|   |   | CAPTURE_CIF                | 352 × 288 interlaced              |
| CAPTURE_CIFK  |   | 352 × 240 even fields only |                                   |
| CAPTURE_QCIF  |   | 176 × 144 even fields only |                                   |
| CAPTURE_SQCIF   |   | 128 × 96 even fields only  |                                   |
| Return Value  | BOOL  | TRUE                       | Success                           |
|   |   | FALSE                      | Failure (check heap size)         |
| Description   | This function installs the video capture subsystem settings and events as follows:  |                            |                                   |
|   | <div><input type="checkbox"/> Creates the buffer structure</div> <div><input type="checkbox"/> Sets up but does not enable video capture events</div> |                            |                                   |
| <div>Note:</div> <div>Before calling Capture_Install(), you must first call Capture_Init(). See page 5-18 for more information on Capture_Init().</div> |   |                            |                                   |

**Example**

```
BOOL B;  
  
Capture_Init();  
  
if (Capture_CardPresent()) {  
    B = Capture_Install(CAPTURE_NTSC, CAPTURE_YUV422,  
                        CAPTURE_CIF);  
  
    if (B) {  
        Capture_Enable();  
        /* ... do main programming ... */  
        Capture_Disable();  
        Capture_UnInstall();  
    }  
}
```



|                      |  |
|----------------------|--|
| <b>Function Name</b> | <b>Capture_InstallSema</b>   |
| <b>Syntax</b>        | long Capture_InstallSema(long SemaId);   |
| <b>Arguments</b>     | long SemaId    ID of an opened semaphore (TaskOpenSema())  |
| <b>Return Value</b>  | long            Old semaphore value  |
| <b>Description</b>   | This function installs the video capture semaphore. The semaphore ID argument must be obtained from calling TaskOpenSema(). The video capture ISR signals this semaphore whenever a new frame of video is ready.   |
| <b>Example</b>       | <pre>long  CaptureSemaId; ULONG Buff;  Capture_Init(); Capture_Install(CAPTURE_NTSC,CAPTURE_RGB888,CAPTURE_640x480); CaptureSemaId = TaskOpenSema(-1,0); <b>Capture_InstallSema(CaptureSemaId);</b> Capture_Enable();  while (1) {     TaskWaitSema(CaptureSemaId);     Buff = Capture_GetBuffer();      /* do processing here */      Capture_FreeBuffer(); }</pre> |

|                      |  |              |   |              |                                   |              |                                   |
|----------------------|--|--------------|---|--------------|-----------------------------------|--------------|-----------------------------------|
| <b>Function Name</b> | <b>Capture_SetInputSource</b>  |              |   |              |                                   |              |                                   |
| <b>Syntax</b>        | void Capture_SetInputSource(BYTE <i>InSrc</i> );   |              |   |              |                                   |              |                                   |
| <b>Arguments</b>     | <p>BYTE <i>InSrc</i> Input source:</p> <table> <tr> <td>CAPTURE_SVHS</td><td>Decode SVHS input from VID1 and VID2 inputs</td></tr> <tr> <td>CAPTURE_VID1</td><td>Decode CVBS input from VID1 input</td></tr> <tr> <td>CAPTURE_VID2</td><td>Decode CVBS input from VID2 input</td></tr> </table>  | CAPTURE_SVHS | Decode SVHS input from VID1 and VID2 inputs | CAPTURE_VID1 | Decode CVBS input from VID1 input | CAPTURE_VID2 | Decode CVBS input from VID2 input |
| CAPTURE_SVHS         | Decode SVHS input from VID1 and VID2 inputs  |              |   |              |                                   |              |                                   |
| CAPTURE_VID1         | Decode CVBS input from VID1 input  |              |   |              |                                   |              |                                   |
| CAPTURE_VID2         | Decode CVBS input from VID2 input  |              |   |              |                                   |              |                                   |
| <b>Return Value</b>  | None   |              |   |              |                                   |              |                                   |
| <b>Description</b>   | <p>This function selects the input source to the video decoder. There are two RCA input jacks labeled VID1 and VID2. A composite input source can be connected to either one of these, or both of them may be used for an S-VHS input. An on-board analog multiplexer takes care of routing, and glue logic (that is, intermediate interface logic) takes care of multiplexing on the high-speed analog-to-digital converters (ADCs).</p> <div> <p><b>Note:</b></p> <p>Before calling Capture_SetInputSource(), you must first call Capture_Init(). See page 5-18 for more information on Capture_Init().</p> </div> |              |   |              |                                   |              |                                   |
| <b>Example</b>       | <pre> BOOL B;  Capture_Init();  if (Capture_CardPresent()) {     B = Capture_Install(CAPTURE_NTSC,CAPTURE_YUV422,CAP- TURE_CIF);      if (B) {         Capture_SetInputSource(CAPTURE_VID1);         Capture_Enable();         /* ... do main programming ... */         Capture_Disable();         Capture_UnInstall();     } } </pre>  |              |   |              |                                   |              |                                   |

|                      |  |  |                            |
|----------------------|--|--|----------------------------|
| <b>Function Name</b> | <b>Capture_SetScaling</b>  |  |                            |
| <b>Syntax</b>        | BOOL Capture_SetScaling(BYTE <i>Scaling</i> );   |  |                            |
| <b>Arguments</b>     | BYTE <i>Scaling</i>  | New scaling value, one of the following: |                            |
|                      |  | CAPTURE_640x480                          | 640 × 480 interlaced       |
|                      |  | CAPTURE_512x512                          | 512 × 512 interlaced       |
|                      |  | CAPTURE_CIF                              | 352 × 288 interlaced       |
|                      |  | CAPTURE_CIFK                             | 352 × 240 even fields only |
|                      |  | CAPTURE_QCIF                             | 176 × 144 even fields only |
|                      |  | CAPTURE_SQCIF                            | 128 × 96 even fields only  |
| <b>Return Value</b>  | BOOL   | TRUE                                     | Success                    |
|                      |  | FALSE                                    | Failure (check heap size)  |
| <b>Description</b>   | <p>This function sets the video capture scaling. When you first call Capture_Install(), you specify a scaling constant. However, if there is a need to change the scaling size dynamically, use this function. This function must not be called when video capture is enabled. This function first frees any buffer memory allocated from the previous settings. New buffers are allocated from the heap to match the new scaling size. An application can call Capture_GetMetrics() after calling this function to get the dimensions and buffer sizes.</p> |  |                            |
| <b>Example</b>       | <pre> long  CaptureSemaId; ULONG Buff;  Capture_Init(); Capture_Install(CAPTURE_NTSC,CAPTURE_RGB888,CAPTURE_640x480); CaptureSemaId = TaskOpenSema(-1,0); Capture_InstallSema(CaptureSemaId); Capture_Enable();  while (1) {     TaskWaitSema(CaptureSemaId);     Buff = Capture_GetBuffer();      /* do processing here */      Capture_FreeBuffer();      if (some condition) {         Capture_Disable();         Capture_SetScaling(CAPTURE_SQCIF);         Capture_Enable();     } } </pre>   |  |                            |

|                      |   |
|----------------------|---|
| <b>Function Name</b> | <b>Capture_UnInstall</b>  |
| <b>Syntax</b>        | <code>void Capture_UnInstall();</code>  |
| <b>Arguments</b>     | None  |
| <b>Return Value</b>  | None  |
| <b>Description</b>   | <p>This function performs the following actions:</p> <ul style="list-style-type: none"><li><input type="checkbox"/> Disables video capture</li><li><input type="checkbox"/> Uninstalls all video capture events that were set up by <code>Capture_Install()</code></li><li><input type="checkbox"/> Frees any memory allocated for buffer storage</li></ul> |
| <b>Example</b>       | <pre>BOOL B;  Capture_Init();  if (Capture_CardPresent()) {     B = Capture_Install(CAPTURE_NTSC, CAPTURE_YUV422,         CAPTURE_CIF);      if (B) {         Capture_SetInputSource(CAPTURE_VID1);         Capture_Enable();         /* ... do main programming ... */         Capture_Disable();         <b>Capture_UnInstall();</b>     } }</pre>        |

# Host Communications API

This chapter discusses the host communications data types and macros. It also describes, in alphabetical order, the application programming interface (API) functions associated with the host communications drivers for the TMS320C8x software development board (SDB).

Two host communications driver libraries exist: one links into a host application and the other links into an SDB application. The host library contains a complete set of primitives used to communicate to the board plus a set of client management functions. The SDB driver contains a set of server management functions. Together, with the host acting as the client and the SDB acting as the server, the two libraries allow a host application to send commands to an SDB application. The libraries do not define the context of the commands—they just provide the means to transmit them and synchronize the two applications. This guide refers to the host library as the *client* library and the corresponding SDB library as the *server* library.

| Topic   | Page |
|---|------|
| 6.1 Host Communications API Macros and Data Types .....         | 6-2  |
| 6.2 Interaction Between the Client API and the Server API ..... | 6-6  |
| 6.3 Bootstrapping the SDB from the Host .....                   | 6-10 |
| 6.4 Host Communications API Functions .....                     | 6-11 |

## 6.1 Host Communications API Macros and Data Types

Table 6–1 describes the macros used by the host communications API and lists the API functions that use each macro. Figure 6–1 provides the definition for the host communications API data type. The macros, data type, and function prototypes for the client API are defined in <hclient.h>. The client API object code resides in hsdbrvs.lib. The macros and the function prototypes for the server API are defined in <sserver.h>. The server API object code resides in sdbdrvs.lib.

Table 6–1. Host Communications API Macros

(a) Client status bits

| Macro Name                    | Value  | Description              |
|-------------------------------|--------|--------------------------|
| #define CLIENT_STATOK         | 0x0000 | Status okay              |
| #define CLIENT_TIMEOUT        | 0x0001 | Time-out occurred        |
| #define CLIENT_DEAD           | 0x0002 | Device did not respond   |
| #define CLIENT_MAILBOXFULL    | 0x0004 | FIFO mailbox was full    |
| #define CLIENT_MAILBOXEMPTY   | 0x0008 | FIFO mailbox was empty   |
| #define CLIENT_FIFOALMOSTFULL | 0x0010 | FIFO was almost full     |
| #define CLIENT_FIFOEMPTY      | 0x0020 | FIFO was empty           |
| #define CLIENT_CLOSED         | 0x0040 | Device was not opened    |
| #define CLIENT_BITSET         | 0x0080 | Bit test result          |
| #define CLIENT_BOOTFILE       | 0x0100 | Could not open boot file |
| #define CLIENT_COFFFILE       | 0x0200 | Could not open COFF file |
| #define CLIENT_BINFILE        | 0x0400 | Could not open bin file  |

**Note:** These macros are used by all client API functions except Client\_Init().

(b) PCI status bits

| Macro Name           | Value      | Bit Position | Description               |
|----------------------|------------|--------------|---------------------------|
| #define CLIENT_MRST  | 0x00000001 | [00]         | Master reset (active low) |
| #define CLIENT_FRST  | 0x00000002 | [01]         | FIFO reset (active low)   |
| #define CLIENT_FSW0  | 0x00000004 | [02]         | Byte swapping 0           |
| #define CLIENT_FSW1  | 0x00000008 | [03]         | Byte swapping 1           |
| #define CLIENT_FOFF0 | 0x00000010 | [04]         | FIFO flag offset 0        |
| #define CLIENT_FOFF1 | 0x00000020 | [05]         | FIFO flag offset 1        |
| #define CLIENT_IAEN  | 0x00000040 | [06]         | Host interrupt enable     |

Table 6–1. Host Communications API Macros (Continued)

| Macro Name          | Value      | Bit Position | Description                   |
|---------------------|------------|--------------|-------------------------------|
| #define CLIENT_BLR  | 0x00000080 | [07]         | Block transfer read           |
| #define CLIENT_BLW  | 0x00000100 | [08]         | Block transfer write          |
| #define CLIENT_BDIS | 0x00000200 | [09]         | Burst disable                 |
| #define CLIENT_R0   | 0x00000400 | [10]         | Reserved                      |
| #define CLIENT_GPO0 | 0x00000800 | [11]         | General purpose output 0      |
| #define CLIENT_GPO1 | 0x00001000 | [12]         | General purpose output 1      |
| #define CLIENT_R1   | 0x00002000 | [13]         | Reserved                      |
| #define CLIENT_R2   | 0x00004000 | [14]         | Reserved                      |
| #define CLIENT_EF2  | 0x00008000 | [15]         | SDB-to-host FIFO empty flag   |
| #define CLIENT_EF1  | 0x00010000 | [16]         | Host-to-SDB FIFO empty flag   |
| #define CLIENT_AF1  | 0x00020000 | [17]         | SDB-to-host almost full flag  |
| #define CLIENT_AF2  | 0x00040000 | [18]         | Host-to-SDB almost full flag  |
| #define CLIENT_MB1  | 0x00080000 | [19]         | Host-to-SDB mailbox full flag |
| #define CLIENT_MB2  | 0x00100000 | [20]         | SDB-to-host mailbox full flag |
| #define CLIENT_PRGD | 0x00200000 | [21]         | EPLD programming done         |
| #define CLIENT_GPI0 | 0x00400000 | [22]         | General purpose input 0       |
| #define CLIENT_GPI1 | 0x00800000 | [23]         | General purpose input 1       |
| #define CLIENT_R3   | 0x01000000 | [24]         | Reserved                      |
| #define CLIENT_R4   | 0x02000000 | [25]         | Reserved                      |
| #define CLIENT_R5   | 0x04000000 | [26]         | Reserved                      |
| #define CLIENT_R6   | 0x08000000 | [27]         | Reserved                      |
| #define CLIENT_R7   | 0x10000000 | [28]         | Reserved                      |
| #define CLIENT_R8   | 0x20000000 | [29]         | Reserved                      |
| #define CLIENT_R9   | 0x40000000 | [30]         | Reserved                      |
| #define CLIENT_R10  | 0x80000000 | [31]         | Reserved                      |

**Note:** These macros are used by the following functions:  
 Client\_ClearConfigBit()  
 Client\_ReadConfigBit()  
 Client\_SetConfigBit()

Table 6–1. Host Communications API Macros (Continued)

## (c) Boot and reset bits

| Macro Name             | Value  | Description                               |
|------------------------|--------|---|
| #define CLIENT_RUN     | 0x0001 | Do not run COFF file <sup>†</sup>         |
| #define CLIENT_LOAD    | 0x0002 | Do not load COFF file <sup>†</sup>        |
| #define CLIENT_EPOINT  | 0x0004 | Custom entry point specified <sup>†</sup> |
| #define CLIENT_TBCOFF  | 0x0008 | Disable TBC emulator chip <sup>‡</sup>    |
| #define CLIENT_EMURST  | 0x0010 | Execute emurst.exe <sup>‡</sup>           |
| #define CLIENT_VERBOSE | 0x0020 | Display messages <sup>§</sup>             |

<sup>†</sup> Macros used by the function Client\_Reset()<sup>‡</sup> Macros used by the function Client\_Boot()<sup>§</sup> Macro used by the functions Client\_Reset() and Client\_Boot()

## (d) FIFO data swapping constants

| Macro Name                   | Value  | Description of Data Swap |
|------------------------------|--------|--------------------------|
| #define CLIENT_NOSWAP        | 0x0000 | 0×12345678 → 0×12345678  |
| #define CLIENT_BYTESWAP      | 0x0004 | 0×12345678 → 0×78563412  |
| #define CLIENT_WORDSWAP      | 0x0008 | 0×12345678 → 0×56781234  |
| #define CLIENT_BYTEWORD-SWAP | 0x000C | 0×12345678 → 0×34127856  |

**Note:** These macros are used by the function Client\_SetSwapping().

## (e) Client command size

| Macro Name               | Value  | Description                         |
|--------------------------|--------|-------------------------------------|
| #define CLIENT_CMND_SIZE | 0x0010 | Number of 32-bit words in a command |

**Note:** This value is used internal to the API and must match SERVER\_CMND\_SIZE.

## (f) Server command size

| Macro Name               | Value  | Description                         |
|--------------------------|--------|-------------------------------------|
| #define SERVER_CMND_SIZE | 0x0010 | Number of 32-bit words in a command |

**Note:** This value is used internal to the API and must match CLIENT\_CMND\_SIZE.



Figure 6–1. Host Communications API Data Type (CLIENT\_STAT)

```

typedef USHORT CLIENT_STAT;

/* CLIENT_STAT returns status information to the application: */
/* Bit */
/* Position  Bit Name      Description */
/* [00]      CLIENT_STATOK  No errors occurred. */
/* [01]      CLIENT_TIMEOUT The access timed out. */
/* [02]      CLIENT_DEAD    The SDB did not respond. */
/* [03]      CLIENT_MAILBOXFULL The time-out was caused by attempting */
/*                               write to a full mailbox. */
/* [04]      CLIENT_MAILBOXEMPTY The time-out was caused by attempting */
/*                               read from an empty mailbox. */
/* [05]      CLIENT_FIFOALMOSTFULL The time-out was caused by attempting */
/*                               write to an almost full FIFO */
/* [06]      CLIENT_FIFOEMPTY  The time-out was caused by attempting */
/*                               read from an empty FIFO. */
/* [07]      CLIENT_CLOSED    Function could not complete because */
/*                               the device has not been opened. */
/* [08]      CLIENT_BITSET    The configuration bit tested was set. */
/* [09]      CLIENT_BOOTFILE  The sdbboot.out file could not be */
/*                               opened. */
/* [10]      CLIENT_COFFFILE  The COFF file to be loaded could not */
/*                               be opened. */
/* [11]      CLIENT_BINFILE   The .bin file could not be opened. */
/* [12]      reserved        This bit is reserved for future use. */
/* [13]      reserved        This bit is reserved for future use. */
/* [14]      reserved        This bit is reserved for future use. */
/* [15]      reserved        This bit is reserved for future use. */

```

## **6.2 Interaction Between the Client API and the Server API**

The client API provides functions for communicating with the SDB via the SDB device driver for Windows. These functions include simple primitives for reading to and writing from the SDB's I/O space and the PCI interface FIFO. Also included in the API is client/server command passing functionality. For the client/server protocol to work, the host has to be running the client API, and the SDB must be running the server API. When the client/server software is running, the host can pass application-defined commands to the SDB.

The API does not define the context of commands, but it does define the structure. A command is nothing more than a sequence of 32-bit words. The number of 32-bit words that make up a command is defined by `CLIENT_CMND_SIZE` for the client API and by `SERVER_CMND_SIZE` for the server API. These two values are the same and, if you alter them, they must always be equal. The first 32-bit word in the command is defined to be the command ID. The remaining 32-bit words may be used however the application wishes. The command size is set by TI to 16. This size provides fifteen 32-bit words for applications to use for argument passing in a command, plus one application-defined command ID. The 'C80 server is an interrupt service routine (ISR) that, when triggered by the host, accepts the command, and then signals the 'C80 application.

### **6.2.1 Client/Server Synchronization**

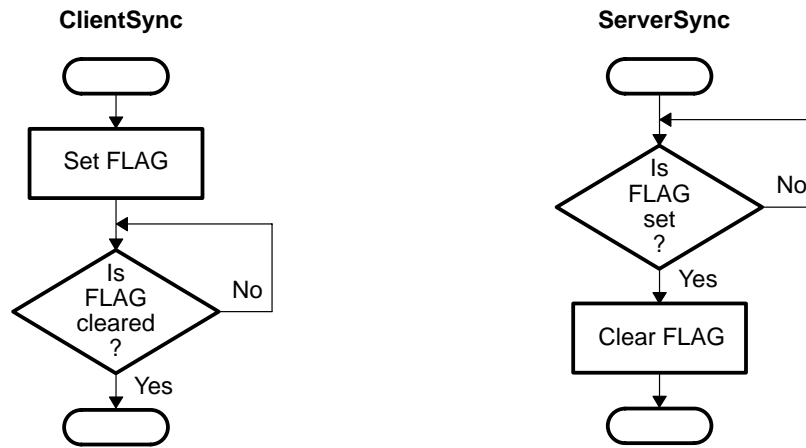
One important aspect of the client/server operation is synchronization. Both the client and server APIs have a synchronization function that ensures events occur when they are supposed to. These synchronization functions are:

- ☐ `Client_Sync()` in the client API for the host
- ☐ `Server_Sync()` in the server API for the SDB

When the `Client_Sync()` function is called, it does not return to the host application until the SDB application calls `Server_Sync()`. When the `Server_Sync()` function is called, it does not return to the SDB application until the host application calls `Client_Sync()`. The result is the `Client_Sync()` function returns to the host application at virtually the same time that the `Server_Sync()` function returns to the SDB application. This mechanism synchronizes the two applications.

Figure 6–2 illustrates, in generic terms, how the `Client_Sync()` function interacts with the `Server_Sync()` function. `FLAG` is a bit visible to both the host and SDB.

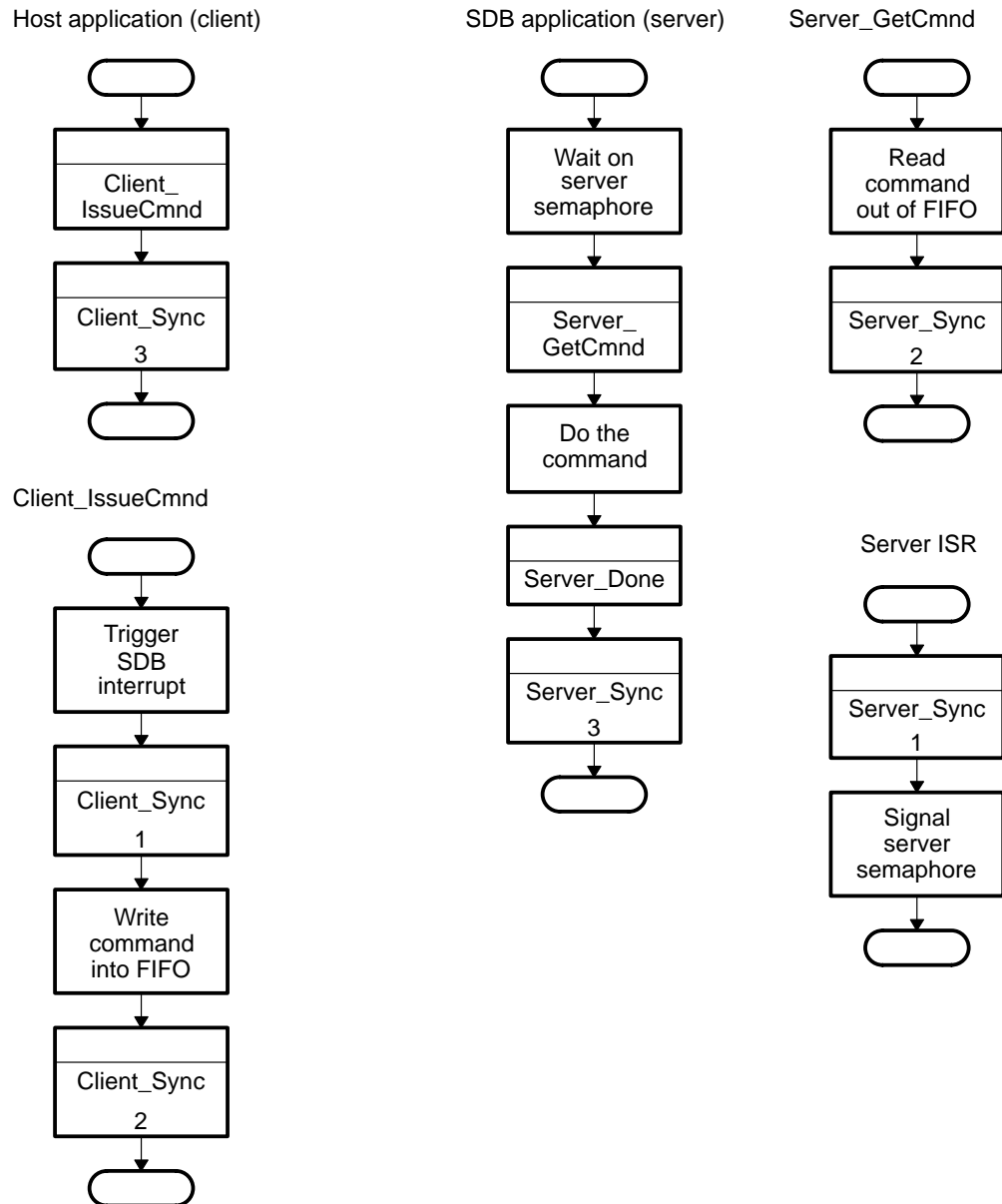
Figure 6–2. Client/Server Synchronization



## **6.2.2 Client to Server Protocol**

Figure 6–3 illustrates the flow of the host client sending a command to the SDB server. Keep in mind that the host and the SDB applications are both running at the same time. The host issues a command that interrupts the SDB. The SDB ISR signals a semaphore, which is received by the SDB application. The SDB application gets the command and acts on it. As seen in the flow charts, synchronization functions have been placed at strategic locations to ensure both applications work in harmony.

Figure 6–3. Client/Server Command Flow



### **6.3 Bootstrapping the SDB from the Host**

A required and fundamental aspect of the SDB is the ability to bootstrap it from the host. Bootstrapping is the process of bringing the 'C80 out of reset and then providing it some code to run. The host can reset the SDB by writing a 0 then a 1 into the CLIENT\_MRST (master reset) bit position of the PCI status register (PCISTAT). The 'C80 and other hardware on the board are reset, but the 'C80 comes out of reset halted. The 'C80 must be unhalted by asserting its  $\overline{\text{EINT3}}$  pin, which can be done by the host. Once unhalted after reset, the 'C80 immediately begins executing the instruction located at address 0xFFFFFFFF8 in the 'C80's address space. Because the 'C80 just came out of reset, its instruction cache is empty, so it must do an instruction cache subblock fill to fetch its first instruction. Therefore, the 'C80 must load 64 bytes into cache. Because all cache subblocks are 64-byte aligned, 64 bytes will be loaded starting at address 0xFFFFF8C0. This address falls in the range of the PCI interface FIFO, implying 64 bytes of instructions must be in the FIFO for the 'C80 to complete its instruction cache fill. The host is responsible for putting this data into the FIFO.

Two methods of bootstrapping the SDB are implemented. The first method loads two 'C80 programs stored in binary format: miniboot.bin and bootserv.bin. The miniboot program is bootstrapped to the 'C80, which executes and loads the bootserv program. These are default programs that reset the board. The host API function Client\_Reset() as well as the board reset utility sdbrst.exe loads these two programs.

The second method of bootstrapping the SDB involves loading a boot server program, which then loads a user-specified common object file format (COFF) file for execution. The host API function Client\_Boot() resets the board, loads a specified COFF file, and executes it. In other words, Client\_Boot() loads and runs a 'C80 program from the host.

Included with the API libraries is a 'C80 application named sdbboot.out (in COFF format). The Client\_Boot() function bootstraps this file to the 'C80, which runs entirely from MP cache. The sdbboot.out file is responsible for accepting a COFF file from the host, storing it in memory on the board, and then executing it. The boot program (sdbboot.out) is written to fit precisely into seven MP instruction cache subblocks. This is accomplished by forcing the 'C80 to do seven instruction cache subblock fills in the same order that the host writes the subblocks into the PCI interface FIFO. The source for sdbboot.out is included with the SDB software so that you can customize it.

## 6.4 Host Communications API Functions

Listed below in alphabetical order are the host communications API functions. Use this list as a table of contents to the host communications API functions.

| <b>Function</b>             | <b>Page</b> | <b>Function</b>            | <b>Page</b> |
|-----------------------------|-------------|----------------------------|-------------|
| Client_Boot .....           | 6-12        | Client_Sync .....          | 6-26        |
| Client_ClearConfigBit ..... | 6-13        | Client_WriteConfig .....   | 6-27        |
| Client_Close .....          | 6-14        | Client_WriteDataFifo ..... | 6-28        |
| Client_Init .....           | 6-15        | Client_Writel0 .....       | 6-29        |
| Client_IssueCmnd .....      | 6-16        | Client_WriteMailbox .....  | 6-30        |
| Client_Open .....           | 6-17        | Server_Done .....          | 6-31        |
| Client_ReadConfig .....     | 6-18        | Server_GetCmnd .....       | 6-32        |
| Client_ReadConfigBit .....  | 6-19        | Server_Init .....          | 6-33        |
| Client_ReadDataFifo .....   | 6-20        | Server_Install .....       | 6-34        |
| Client_Readl0 .....         | 6-21        | Server_InstallSema .....   | 6-35        |
| Client_ReadMailbox .....    | 6-22        | Server_ReadDataFifo .....  | 6-36        |
| Client_Reset .....          | 6-23        | Server_Sync .....          | 6-37        |
| Client_SetConfigBit .....   | 6-24        | Server_UnInstall .....     | 6-38        |
| Client_SetSwapping .....    | 6-25        | Server_WriteDataFifo ..... | 6-39        |

|                               |   |                             |  |                               |                                    |                           |   |                             |                         |                            |                                    |                          |                  |                         |                       |                           |                                    |
|-------------------------------|---|-----------------------------|--|-------------------------------|------------------------------------|---------------------------|---|-----------------------------|-------------------------|----------------------------|------------------------------------|--------------------------|------------------|-------------------------|-----------------------|---------------------------|------------------------------------|
| <b>Function Name</b>          | <b>Client_Boot</b>  |                             |  |                               |                                    |                           |   |                             |                         |                            |                                    |                          |                  |                         |                       |                           |                                    |
| <b>Syntax</b>                 | <code>CLIENT_STAT Client_Boot(char *coffname, ULONG EntryPoint, USHORT Flags, FILE *Output);</code>   |                             |  |                               |                                    |                           |   |                             |                         |                            |                                    |                          |                  |                         |                       |                           |                                    |
| <b>Arguments</b>              | <table><tr><td><code>char *coffname</code></td><td>COFF filename (may include path information)</td></tr><tr><td><code>ULONG EntryPoint</code></td><td>Alternative program entry point</td></tr><tr><td><code>USHORT Flags</code></td><td><table><tr><td><code>CLIENT_VERBOSE</code></td><td>Send messages to Output</td></tr><tr><td><code>CLIENT_EPOINT</code></td><td>Address of alternative entry point</td></tr><tr><td><code>CLIENT_LOAD</code></td><td>Do the COFF load</td></tr><tr><td><code>CLIENT_RUN</code></td><td>Execute the COFF file</td></tr></table></td></tr><tr><td><code>FILE *Output</code></td><td>Opened text stream, such as stdout</td></tr></table>   | <code>char *coffname</code> | COFF filename (may include path information) | <code>ULONG EntryPoint</code> | Alternative program entry point    | <code>USHORT Flags</code> | <table><tr><td><code>CLIENT_VERBOSE</code></td><td>Send messages to Output</td></tr><tr><td><code>CLIENT_EPOINT</code></td><td>Address of alternative entry point</td></tr><tr><td><code>CLIENT_LOAD</code></td><td>Do the COFF load</td></tr><tr><td><code>CLIENT_RUN</code></td><td>Execute the COFF file</td></tr></table> | <code>CLIENT_VERBOSE</code> | Send messages to Output | <code>CLIENT_EPOINT</code> | Address of alternative entry point | <code>CLIENT_LOAD</code> | Do the COFF load | <code>CLIENT_RUN</code> | Execute the COFF file | <code>FILE *Output</code> | Opened text stream, such as stdout |
| <code>char *coffname</code>   | COFF filename (may include path information)  |                             |  |                               |                                    |                           |   |                             |                         |                            |                                    |                          |                  |                         |                       |                           |                                    |
| <code>ULONG EntryPoint</code> | Alternative program entry point   |                             |  |                               |                                    |                           |   |                             |                         |                            |                                    |                          |                  |                         |                       |                           |                                    |
| <code>USHORT Flags</code>     | <table><tr><td><code>CLIENT_VERBOSE</code></td><td>Send messages to Output</td></tr><tr><td><code>CLIENT_EPOINT</code></td><td>Address of alternative entry point</td></tr><tr><td><code>CLIENT_LOAD</code></td><td>Do the COFF load</td></tr><tr><td><code>CLIENT_RUN</code></td><td>Execute the COFF file</td></tr></table>   | <code>CLIENT_VERBOSE</code> | Send messages to Output                      | <code>CLIENT_EPOINT</code>    | Address of alternative entry point | <code>CLIENT_LOAD</code>  | Do the COFF load  | <code>CLIENT_RUN</code>     | Execute the COFF file   |                            |                                    |                          |                  |                         |                       |                           |                                    |
| <code>CLIENT_VERBOSE</code>   | Send messages to Output   |                             |  |                               |                                    |                           |   |                             |                         |                            |                                    |                          |                  |                         |                       |                           |                                    |
| <code>CLIENT_EPOINT</code>    | Address of alternative entry point  |                             |  |                               |                                    |                           |   |                             |                         |                            |                                    |                          |                  |                         |                       |                           |                                    |
| <code>CLIENT_LOAD</code>      | Do the COFF load  |                             |  |                               |                                    |                           |   |                             |                         |                            |                                    |                          |                  |                         |                       |                           |                                    |
| <code>CLIENT_RUN</code>       | Execute the COFF file   |                             |  |                               |                                    |                           |   |                             |                         |                            |                                    |                          |                  |                         |                       |                           |                                    |
| <code>FILE *Output</code>     | Opened text stream, such as stdout  |                             |  |                               |                                    |                           |   |                             |                         |                            |                                    |                          |                  |                         |                       |                           |                                    |
| <b>Return Value</b>           | <code>CLIENT_STAT</code> Status information   |                             |  |                               |                                    |                           |   |                             |                         |                            |                                    |                          |                  |                         |                       |                           |                                    |
| <b>Description</b>            | <p>This function bootstraps the SDB by resetting the board, loading the specified COFF file, and then executing it. Basically, this function loads and runs a 'C80 program. The function looks in the current directory for the COFF file. If an entry point is specified, program execution begins at that point rather than the default address in the COFF file.</p> <p>Four flags can be set to alter the functionality. Setting the <code>CLIENT_VERBOSE</code> flag instructs the function to output text messages to the stream pointed to by the Output argument. If you set the <code>CLIENT_EPOINT</code> flag, the EntryPoint argument is used; otherwise, the argument is ignored. Setting <code>CLIENT_LOAD</code> instructs the function to load the COFF file. Setting <code>CLIENT_RUN</code> instructs the function to execute the COFF file.</p> <p>Sometimes it may be desirable to load a COFF file but execute it at a later time. To do this, first call this function with the load flag set but the run flag cleared. Then, call this function later with the run flag set and the load flag cleared. This works fine as long as the SDB is not disrupted between calls (from a reset, for instance). If the <code>sdbboot.out</code> file cannot be opened, the <code>CLIENT_BOOTFILE</code> flag is set in the return status. If the COFF file cannot be opened, the <code>CLIENT_COFFFILE</code> flag is set in the return status.</p> |                             |  |                               |                                    |                           |   |                             |                         |                            |                                    |                          |                  |                         |                       |                           |                                    |
| <b>Example</b>                | <pre>CLIENT_STAT St;  St = Client_Boot("myprog.out", NULL, CLIENT_RUN CLIENT_LOAD      CLIENT_VERBOSE, stdout);  if (St == CLIENT_STATOK) {     /* boot went okay */ }</pre>  |                             |  |                               |                                    |                           |   |                             |                         |                            |                                    |                          |                  |                         |                       |                           |                                    |



|                      |   |                        |                                    |
|----------------------|---|------------------------|------------------------------------|
| <b>Function Name</b> | <b>Client_ClearConfigBit</b>  |                        |                                    |
| <b>Syntax</b>        | CLIENT_STAT Client_ClearConfigBit(ULONG <i>bit</i> );   |                        |                                    |
| <b>Arguments</b>     | ULONG <i>bit</i>  | Bit position constant: |                                    |
|                      |   | CLIENT_MRST            | [00] Master reset (active low)     |
|                      |   | CLIENT_FRST            | [01] FIFO reset (active low)       |
|                      |   | CLIENT_FSW0            | [02] Byte swapping 0               |
|                      |   | CLIENT_FSW1            | [03] Byte swapping 1               |
|                      |   | CLIENT_FOFF0           | [04] FIFO flag offset 0            |
|                      |   | CLIENT_FOFF1           | [05] FIFO flag offset 1            |
|                      |   | CLIENT_IAEN            | [06] Host Interrupt enable         |
|                      |   | CLIENT_BLR             | [07] Block transfer read           |
|                      |   | CLIENT_BLW             | [08] Block transfer write          |
|                      |   | CLIENT_BDIS            | [09] Burst disable                 |
|                      |   | CLIENT_R0              | [10] Reserved                      |
|                      |   | CLIENT_GPO0            | [11] General purpose output 0      |
|                      |   | CLIENT_GPO1            | [12] General purpose output 1      |
|                      |   | CLIENT_R1              | [13] Reserved                      |
|                      |   | CLIENT_R2              | [14] Reserved                      |
|                      |   | CLIENT_EF2             | [15] SDB-to-host FIFO empty flag   |
|                      |   | CLIENT_EF1             | [16] Host-to-SDB FIFO empty flag   |
|                      |   | CLIENT_AF1             | [17] SDB-to-host almost full flag  |
|                      |   | CLIENT_AF2             | [18] Host-to-SDB almost full flag  |
|                      |   | CLIENT_MB1             | [19] Host-to-SDB mailbox full flag |
|                      |   | CLIENT_MB2             | [20] SDB-to-host mailbox full flag |
|                      |   | CLIENT_PRGD            | [21] EPLD programming done         |
|                      |   | CLIENT_GPI0            | [22] General purpose input 0       |
|                      |   | CLIENT_GPI1            | [23] General purpose input 1       |
|                      |   | CLIENT_R3              | [24] Reserved                      |
|                      |   | CLIENT_R4              | [25] Reserved                      |
|                      |   | CLIENT_R5              | [26] Reserved                      |
|                      |   | CLIENT_R6              | [27] Reserved                      |
|                      |   | CLIENT_R7              | [28] Reserved                      |
|                      |   | CLIENT_R8              | [29] Reserved                      |
|                      |   | CLIENT_R9              | [30] Reserved                      |
|                      |   | CLIENT_R10             | [31] Reserved                      |
| <b>Return Value</b>  | CLIENT_STAT Status information  |                        |                                    |
| <b>Description</b>   | This function clears a bit in the PCI status register (PCISTAT).  |                        |                                    |
| <b>Example</b>       | <pre>CLIENT_STAT St;  St = Client_ClearConfigBit(CLIENT_FSW0);  if (St == CLIENT_STATOK) {     /* ... okay ... */ }</pre> |                        |                                    |

|                      |  |
|----------------------|--|
| <b>Function Name</b> | <b>Client_Close</b>  |
| <b>Syntax</b>        | CLIENT_STAT Client_Close();  |
| <b>Arguments</b>     | None   |
| <b>Return Value</b>  | CLIENT_STAT Status information   |
| <b>Description</b>   | This function closes the SDB device. Once it is closed, no other accesses can occur on the SDB until it is reopened via Client_Open(). |
| <b>Example</b>       | <pre>CLIENT_STAT st;  st = Client_Open();  if (st) {     /* ... do some processing ... */     Client_Close(); }</pre>                  |

|  |  |       |                          |
|--|--|-------|--------------------------|
| Function Name  | Client_Init  |       |                          |
| Syntax   | BOOL Client_Init();  |       |                          |
| Arguments  | None   |       |                          |
| Return Value   | BOOL   | TRUE  | Initialization succeeded |
|  |  | FALSE | Initialization failed    |
| Description  | This function initializes the host API. It returns TRUE if the initialization succeeded.                           |       |                          |
| <div><b>Notes:</b><br/>1) You must call Client_Init() <i>before</i> calling any other host API function.<br/>2) You must call this function only once in an application.</div> |  |       |                          |
| Example  | <pre>BOOL ok;<br/><br/>ok = Client_Init();<br/><br/>if (ok) {<br/>    /* ... do some processing ... */<br/>}</pre> |       |                          |

|                      |   |
|----------------------|---|
| <b>Function Name</b> | <b>Client_IssueCmnd</b>   |
| <b>Syntax</b>        | CLIENT_STAT Client_IssueCmnd(ULONG *Cmnd, long <i>TimeOut</i> );  |
| <b>Arguments</b>     | <p>ULONG *Cmnd    Pointer to command; must point to CLIENT_CMND_SIZE<br/>                 32-bit words of preallocated memory</p> <p>long <i>TimeOut</i>    Number of attempts before timing out</p>  |
| <b>Return Value</b>  | CLIENT_STAT    Status information   |
| <b>Description</b>   | <p>This function is the host command client responsible for sending a command to the SDB. Success requires the 'C80 command server to be running also. The context of the command is application defined.</p>   |
| <b>Example</b>       | <pre>#define MY_CMND_ID 0x80120001  void MyClientFunc() {     ULONG Cmnd[CLIENT_CMND_SIZE];     Cmnd[0] = MY_CMND_ID;     Cmnd[1] = argument1;     Cmnd[2] = argument2;     /* ... etc ... */     Client_IssueCmnd(1000);     Client_Sync(100);     /* ... finish up command ... */ }</pre> |

|                      |   |
|----------------------|---|
| <b>Function Name</b> | <b>Client_Open</b>  |
| <b>Syntax</b>        | CLIENT_STAT Client_Open();  |
| <b>Arguments</b>     | None  |
| <b>Return Value</b>  | CLIENT_STAT    Status information   |
| <b>Description</b>   | <p>This function opens the SDB device. Before accesses to the SDB can occur, it must be opened. If the SDB device opens successfully, the return status equals CLIENT_STATOK.</p> <div><p><b>Note:</b> You must call Client_Open():</p><ul style="list-style-type: none"><li><input type="checkbox"/> <i>after</i> calling Client_Init() (see page 6-15 for more information)</li><li><input type="checkbox"/> <i>before</i> calling any other host API functions</li></ul></div> |
| <b>Example</b>       | <pre>CLIENT_STAT st;  if (Client_Init()) {     <b>st = Client_Open();</b>     if (st == CLIENT_STATOK) {         /* ... do some processing ... */         Client_Close();     } }</pre>   |

|                      |   |
|----------------------|---|
| <b>Function Name</b> | <b>Client_ReadConfig</b>  |
| <b>Syntax</b>        | CLIENT_STAT Client_ReadConfig(ULONG *Val);  |
| <b>Arguments</b>     | ULONG *Val/    Pointer to ULONG, which receives the PCI status register contents                      |
| <b>Return Value</b>  | CLIENT_STAT    Status information   |
| <b>Description</b>   | This function reads the PCI status register (PCISTAT) and stores it into the ULONG pointed to by Val. |
| <b>Example</b>       | <pre>ULONG PciStat;<br/><br/>Client_ReadConfig(&amp;PciStat);</pre>                                   |

|                      |   |                        |                               |
|----------------------|---|------------------------|-------------------------------|
| <b>Function Name</b> | <b>Client_ReadConfigBit</b>   |                        |                               |
| <b>Syntax</b>        | CLIENT_STAT Client_ReadConfigBit(ULONG <i>bit</i> );  |                        |                               |
| <b>Arguments</b>     | ULONG <i>bit</i>  | Bit position constant: |                               |
|                      | CLIENT_MRST   | [00]                   | Master reset (active low)     |
|                      | CLIENT_FRST   | [01]                   | FIFO reset (active low)       |
|                      | CLIENT_FSW0   | [02]                   | Byte swapping 0               |
|                      | CLIENT_FSW1   | [03]                   | Byte swapping 1               |
|                      | CLIENT_FOFF0  | [04]                   | FIFO flag offset 0            |
|                      | CLIENT_FOFF1  | [05]                   | FIFO flag offset 1            |
|                      | CLIENT_IAEN   | [06]                   | Host interrupt enable         |
|                      | CLIENT_BLR  | [07]                   | Block transfer read           |
|                      | CLIENT_BLW  | [08]                   | Block transfer write          |
|                      | CLIENT_BDIS   | [09]                   | Burst disable                 |
|                      | CLIENT_R0   | [10]                   | Reserved                      |
|                      | CLIENT_GPO0   | [11]                   | General purpose output 0      |
|                      | CLIENT_GPO1   | [12]                   | General purpose output 1      |
|                      | CLIENT_R1   | [13]                   | Reserved                      |
|                      | CLIENT_R2   | [14]                   | Reserved                      |
|                      | CLIENT_EF2  | [15]                   | SDB-to-host FIFO empty flag   |
|                      | CLIENT_EF1  | [16]                   | Host-to-SDB FIFO empty flag   |
|                      | CLIENT_AF1  | [17]                   | SDB-to-host almost full flag  |
|                      | CLIENT_AF2  | [18]                   | Host-to-SDB almost full flag  |
|                      | CLIENT_MB1  | [19]                   | Host-to-SDB mailbox full flag |
|                      | CLIENT_MB2  | [20]                   | SDB-to-host mailbox full flag |
|                      | CLIENT_PRGD   | [21]                   | EPLD programming done         |
|                      | CLIENT_GPI0   | [22]                   | General purpose input 0       |
|                      | CLIENT_GPI1   | [23]                   | General purpose input 1       |
|                      | CLIENT_R3   | [24]                   | Reserved                      |
|                      | CLIENT_R4   | [25]                   | Reserved                      |
|                      | CLIENT_R5   | [26]                   | Reserved                      |
|                      | CLIENT_R6   | [27]                   | Reserved                      |
|                      | CLIENT_R7   | [28]                   | Reserved                      |
|                      | CLIENT_R8   | [29]                   | Reserved                      |
|                      | CLIENT_R9   | [30]                   | Reserved                      |
|                      | CLIENT_R10  | [31]                   | Reserved                      |
| <b>Return Value</b>  | CLIENT_STAT Status information  |                        |                               |
| <b>Description</b>   | This function reads the PCI status register (PCISTAT) and tests the specified bit. The CLIENT_BITSET flag in the return status indicates whether the tested bit is set or not. CLIENT_BITSET is set if the tested bit is set. |                        |                               |
| <b>Example</b>       | <pre>CLIENT_STAT st;  st = Client_ReadConfigBit(CLIENT_P2H_FEF);  if (st &amp; CLIENT_BITSET) {     /* ... bit was set ... */ }</pre>   |                        |                               |

|               |  |
|---------------|--|
| Function Name | <b>Client_ReadDataFifo</b>   |
| Syntax        | CLIENT_STAT Client_ReadDataFifo(ULONG *Block, ULONG ct, long timeout);   |
| Arguments     | <div>ULONG *Block    Pointer to a block of 32-bit words</div> <div>ULONG ct        Number of words in a block (0 &lt; ct &lt; 8192)</div> <div>long timeout    Number of attempts before timing out</div>  |
| Return Value  | CLIENT_STAT    Status information  |
| Description   | <p>This function reads a block of 32-bit words from the PCI interface FIFO. To prevent reading an empty FIFO, a software watchdog timer is implemented. This function checks the FIFO empty flag and, if it is not set, reads <i>ct</i> 32-bit words and stores them into *Block. If the flag is set, indicating the FIFO is empty, the function loops and tries again. If the function fails <i>timeout</i> times, it returns without reading data and the CLIENT_TIMEOUT flag plus the CLIENT_FIFOEMPTY flag is set in the return status.</p> <div><b>Note:</b><br/>Client_ReadDataFifo() only checks the FIFO empty flag initially. If the block size is larger than the FIFO size, the 'C80 <i>must</i> write the entire block to the FIFO for this function to read out. If this function tries to read more data out of the FIFO than the 'C80 writes, the PCI bus will try to read it out forever, literally.</div> |
| Example       | <pre>ULONG Data[20];<br/><br/>Client_ReadDataFifo(Data,20,100);</pre>  |



|                      |   |   |
|----------------------|---|---|
| <b>Function Name</b> | <b>Client_ReadIo</b>  |   |
| <b>Syntax</b>        | CLIENT_STAT Client_ReadIo(USHORT <i>Address</i> , ULONG <i>*Val</i> );  |   |
| <b>Arguments</b>     | USHORT <i>Address</i>   | Host-relative I/O address   |
|                      | ULONG <i>*Val</i>   | Pointer to ULONG, which receives the PCI status register contents |
| <b>Return Value</b>  | CLIENT_STAT   | Status information  |
| <b>Description</b>   | <p>This function reads a value from the SDB's I/O bus. The address is relative to the host address window into the SDB device. For instance, the MASKEN0 register of the SDB's interrupt controller has a host-relative address of 0x2300. The I/O bus on the SDB is 16 bits wide, but all PCI accesses must be 32-bit accesses. For this reason, the application should mask off the unneeded bits, which contain unpredictable values.</p> <div><b>Note:</b><br/>This function performs an I/O access that requires the use of both PCI interface FIFO mailboxes. If either or both mailboxes are full, the access will fail and the PCI status register (PCISTAT) is returned.</div> |   |
| <b>Example</b>       | <pre>ULONG MaskEn0;<br/><br/>Client_ReadIo(0x2300, &amp;MaskEn0);<br/><br/>MaskEn0 &amp;= 0x0000FFFF;</pre>   |   |

|                      |   |
|----------------------|---|
| <b>Function Name</b> | <b>Client_ReadMailbox</b>   |
| <b>Syntax</b>        | CLIENT_STAT Client_ReadMailbox(ULONG *Val, long <i>timeout</i> );   |
| <b>Arguments</b>     | <div>ULONG *Val    Pointer to ULONG, which receives the PCI status register (PCISTAT) contents</div> <div>long <i>timeout</i>    Number of attempts before timing out</div>   |
| <b>Return Value</b>  | CLIENT_STAT    Status information   |
| <b>Description</b>   | <p>This function reads the 32-bit word from the PCI interface FIFO mailbox. To prevent reading an empty mailbox, a software watchdog timer is implemented. This function checks the mailbox full flag and, if it is set, reads a 32-bit word and stores it in *Val. If the flag is not set, indicating the mailbox is empty, the function loops and tries again. If the function fails <i>timeout</i> times, it returns without reading data, 0x00000000 is written to *Val, and the CLIENT_TIMEOUT flag plus the CLIENT_MAILBOXEMPTY flag is set in the return status.</p> |
| <b>Example</b>       | <pre>ULONG Data;<br/><br/>Client_ReadMailbox(&amp;Data, 100);</pre>   |

|                      |  |                |                         |               |   |               |                    |
|----------------------|--|----------------|-------------------------|---------------|---|---------------|--------------------|
| <b>Function Name</b> | <b>Client_Reset</b>  |                |                         |               |   |               |                    |
| <b>Syntax</b>        | <code>CLIENT_STAT Client_Reset(USHORT <i>Flags</i>, FILE *<i>Output</i>);</code>   |                |                         |               |   |               |                    |
| <b>Arguments</b>     | <p>USHORT <i>Flags</i></p> <table> <tr> <td>CLIENT_VERBOSE</td><td>Send messages to Output</td></tr> <tr> <td>CLIENT_TBCOFF</td><td>Disable test bus controller (TBC) emulator chip</td></tr> <tr> <td>CLIENT_EMURST</td><td>Execute emurst.exe</td></tr> </table> <p>FILE *<i>Output</i>    Opened text stream, such as stdout</p>  | CLIENT_VERBOSE | Send messages to Output | CLIENT_TBCOFF | Disable test bus controller (TBC) emulator chip | CLIENT_EMURST | Execute emurst.exe |
| CLIENT_VERBOSE       | Send messages to Output  |                |                         |               |   |               |                    |
| CLIENT_TBCOFF        | Disable test bus controller (TBC) emulator chip  |                |                         |               |   |               |                    |
| CLIENT_EMURST        | Execute emurst.exe   |                |                         |               |   |               |                    |
| <b>Return Value</b>  | CLIENT_STAT    Status information  |                |                         |               |   |               |                    |
| <b>Description</b>   | <p>This function bootstraps the SDB by resetting the board, loading miniboot.bin, and then loading and running bootserv.bin. The files miniboot.bin and bootserv.bin are the default bootstrapping files and are located in the <code>system32</code> directory of the host operating system.</p> <p>Three flags can be passed to this function:</p> <ul style="list-style-type: none"> <li><input type="checkbox"/> CLIENT_VERBOSE</li> <li><input type="checkbox"/> CLIENT_TBCOFF</li> <li><input type="checkbox"/> CLIENT_EMURST</li> </ul> <p>Setting CLIENT_VERBOSE causes this function to output text messages to the Output stream. Setting CLIENT_TBCOFF causes the on-board TBC emulator chip to be disabled. Disabling this chip allows debugging using the JTAG connector on the board and an XDS510 emulator. Setting CLIENT_EMURST causes this function to execute the emurst.exe utility before resetting. This only applies if you have an XDS510 installed in your PC. By performing an emurst, you can reset the XDS510 and the SDB. The emurst.exe program is not spawned, but rather it is called out as a system command, so the program only needs to be in the system path.</p> |                |                         |               |   |               |                    |
| <b>Example</b>       | <pre>CLIENT_STAT St;  St = Client_Reset(CLIENT_VERBOSE, stdout);  if (St == CLIENT_STATOK) {     /* reset went okay */ }</pre>   |                |                         |               |   |               |                    |

**Function Name****Client\_SetConfigBit****Syntax**CLIENT\_STAT Client\_SetConfigBit(ULONG *bit*);**Arguments**ULONG *bit*

Bit position constant:

|              |      |                               |
|--------------|------|-------------------------------|
| CLIENT_MRST  | [00] | Master reset (active low)     |
| CLIENT_FRST  | [01] | FIFO reset (active low)       |
| CLIENT_FSW0  | [02] | Byte swapping 0               |
| CLIENT_FSW1  | [03] | Byte swapping 1               |
| CLIENT_FOFF0 | [04] | FIFO flag offset 0            |
| CLIENT_FOFF1 | [05] | FIFO flag offset 1            |
| CLIENT_IAEN  | [06] | Host interrupt enable         |
| CLIENT_BLR   | [07] | Block transfer read           |
| CLIENT_BLW   | [08] | Block transfer write          |
| CLIENT_BDIS  | [09] | Burst disable                 |
| CLIENT_R0    | [10] | Reserved                      |
| CLIENT_GPO0  | [11] | General purpose output 0      |
| CLIENT_GPO1  | [12] | General purpose output 1      |
| CLIENT_R1    | [13] | Reserved                      |
| CLIENT_R2    | [14] | Reserved                      |
| CLIENT_EF2   | [15] | SDB-to-host FIFO empty flag   |
| CLIENT_EF1   | [16] | Host-to-SDB FIFO empty flag   |
| CLIENT_AF1   | [17] | SDB-to-host almost full flag  |
| CLIENT_AF2   | [18] | Host-to-SDB almost full flag  |
| CLIENT_MB1   | [19] | Host-to-SDB mailbox full flag |
| CLIENT_MB2   | [20] | SDB-to-host mailbox full flag |
| CLIENT_PRGD  | [21] | EPLD programming done         |
| CLIENT_GPI0  | [22] | General purpose input 0       |
| CLIENT_GPI1  | [23] | General purpose input 1       |
| CLIENT_R3    | [24] | Reserved                      |
| CLIENT_R4    | [25] | Reserved                      |
| CLIENT_R5    | [26] | Reserved                      |
| CLIENT_R6    | [27] | Reserved                      |
| CLIENT_R7    | [28] | Reserved                      |
| CLIENT_R8    | [29] | Reserved                      |
| CLIENT_R9    | [30] | Reserved                      |
| CLIENT_R10   | [31] | Reserved                      |

**Return Value**

CLIENT\_STAT Status information

**Description**

This function sets a bit in the PCI status register (PCISTAT).

**Example**

```
CLIENT_STAT St;

St = Client_SetConfigBit(CLIENT_FSW0);

if (St == CLIENT_STATOK) {
    /* status = okay */
}
```

|               |   |                      |                         |
|---------------|---|----------------------|-------------------------|
| Function Name | Client_SetSwapping  |                      |                         |
| Syntax        | CLIENT_STAT Client_SetSwapping(USHORT sw);  |                      |                         |
| Arguments     | USHORT sw   | CLIENT_NOSWAP        | 0x12345678 → 0x12345678 |
|               |   | CLIENT_BYTESWAP      | 0x12345678 → 0x78563412 |
|               |   | CLIENT_WORDSWAP      | 0x12345678 → 0x56781234 |
|               |   | CLIENT_BYTEWORD-SWAP | 0x12345678 → 0x34127856 |
| Return Value  | CLIENT_STAT Status information  |                      |                         |
| Description   | <p>This function configures the PCI interface FIFO byte swapping logic. The byte swapping logic of the FIFO interface rearranges each 32-bit value that passes through the data FIFO. The swapping logic affects the host-to-SDB FIFO and the SDB-to-host FIFO. It does not affect the FIFO mailboxes (therefore I/O accesses are not affected by swapping).</p> <div><p><b>Note:</b></p><p>If you change the FIFO swapping logic (to do a transfer, for instance), make sure you restore it to CLIENT_NOSWAPPING because the client/server routines require it this way.</p></div> |                      |                         |
| Example       | <pre>CLIENT_STAT St;<br/><br/>St = Client_SetSwapping(CLIENT_NOSWAP);</pre>   |                      |                         |

|                      |  |
|----------------------|--|
| <b>Function Name</b> | <b>Client_Sync</b>   |
| <b>Syntax</b>        | CLIENT_STAT Client_Sync(long <i>timeout</i> );   |
| <b>Arguments</b>     | long <i>timeout</i> Number of attempts before timing out   |
| <b>Return Value</b>  | CLIENT_STAT     Status information   |
| <b>Description</b>   | When you call this function, it waits until the 'C80 executes Server_Sync() before returning. This synchronizes the host client with the 'C80 server.  |
| <b>Example</b>       | <pre>#define MY_CMND_ID 0x80120001  void MyClientFunc() {     ULONG Cmnd[CLIENT_CMND_SIZE];     Cmnd[0] = MY_CMND_ID;     Cmnd[1] = <i>argument1</i>;     Cmnd[2] = <i>argument2</i>;     /* ... etc ... */     Client_IssueCmnd(1000);     <b>Client_Sync(100);</b>     /* ... finish up command ... */ }</pre> |

|                      |   |
|----------------------|---|
| <b>Function Name</b> | <b>Client_WriteConfig</b>   |
| <b>Syntax</b>        | CLIENT_STAT Client_WriteConfig(ULONG Val);                              |
| <b>Arguments</b>     | ULONG Val     ULONG value to be written to the PCI status register      |
| <b>Return Value</b>  | CLIENT_STAT   Status information  |
| <b>Description</b>   | This function writes Val to the PCI status register (PCISTAT).          |
| <b>Example</b>       | <pre>ULONG PciStat = 0x00000023;<br/>Client_WriteConfig(PciStat);</pre> |

|                      |  |
|----------------------|--|
| <b>Function Name</b> | <b>Client_WriteDataFifo</b>  |
| <b>Syntax</b>        | CLIENT_STAT Client_WriteDataFifo(ULONG *Block, ULONG ct, long timeout);  |
| <b>Arguments</b>     | <div>ULONG *Block     Pointer to a block of 32-bit words</div> <div>ULONG ct         Number of words in a block (0 &lt; ct &lt; 8192)</div> <div>long timeout      Number of attempts before timing out</div>  |
| <b>Return Value</b>  | CLIENT_STAT     Status information   |
| <b>Description</b>   | <p>This function writes a block of 32-bit words to the PCI interface FIFO. To prevent writing to a full FIFO, a software watchdog timer is implemented. This function checks the FIFO almost full flag and, if it is not set, writes <i>ct</i> 32-bit words from *Block into the FIFO. If the flag is set, indicating the FIFO is almost full, the function loops and tries again. If the function fails <i>timeout</i> times, it returns without writing any data and the CLIENT_TIMEOUT flag plus the CLIENT_FIFOALMOSTFULL flag is set in the return status.</p> <div><b>Note:</b><br/>Client_WriteDataFifo() only checks the FIFO almost full flag initially. If the block size is larger than the FIFO size, the 'C80 <i>must</i> read the entire block out of the FIFO. If this function tries to write out more data than the 'C80 reads, the FIFO will become full and the PCI bus will try to write to it forever, literally.</div> |
| <b>Example</b>       | <pre>ULONG Data[20];<br/><br/>Client_WriteDataFifo(Data,20,100);</pre>   |



|                      |   |
|----------------------|---|
| <b>Function Name</b> | <b>Client_WriteIo</b>   |
| <b>Syntax</b>        | CLIENT_STAT Client_WriteIo(USHORT <i>Address</i> , ULONG <i>Val</i> );  |
| <b>Arguments</b>     | <div>ULONG <i>Address</i>    Host-relative I/O address</div> <div>ULONG <i>Val</i>        Value to write to I/O bus</div>   |
| <b>Return Value</b>  | CLIENT_STAT      Status information   |
| <b>Description</b>   | <p>This function writes a value to the SDB's I/O bus. The address is relative to the host address window into the SDB device. For instance, the MASKEN0 register of the SDB's interrupt controller has a host-relative address of 0x2300. The I/O bus on the SDB is 16 bits wide, but all PCI accesses must be 32-bit accesses. Only the lower 16 bits are used.</p> <div><b>Note:</b><br/>This function performs an I/O access that requires the use of both PCI interface FIFO mailboxes. If either or both mailboxes are full, the access will fail.</div> |
| <b>Example</b>       | <pre>Client_WriteIo(0x2300, 0x0800);</pre>  |

|                      |  |   |
|----------------------|--|---|
| <b>Function Name</b> | <b>Client_WriteMailbox</b>   |   |
| <b>Syntax</b>        | CLIENT_STAT Client_WriteMailbox(ULONG <i>Val</i> , long <i>timeout</i> );  |   |
| <b>Arguments</b>     | ULONG <i>Val</i>   | Value to be written to the FIFO mailbox |
|                      | long <i>timeout</i>  | Number of attempts before timing out    |
| <b>Return Value</b>  | CLIENT_STAT  | Status information                      |
| <b>Description</b>   | <p>This function writes a single 32-bit word to the PCI interface mailbox. To prevent writing to a full mailbox, a software watchdog timer is implemented. This function checks the mailbox full flag and, if it is not set, writes <i>Val</i> to the mailbox. If the flag is set, indicating the mailbox is full, the function loops and tries again. If the function fails <i>timeout</i> times, it returns without writing data and the CLIENT_TIMEOUT flag plus the CLIENT_MAILBOXFULL flag is set in the return status.</p> |   |
| <b>Example</b>       | <pre>Client_WriteMailbox(0x12345678,100);</pre>  |   |

|                      |   |
|----------------------|---|
| <b>Function Name</b> | <b>Server_Done</b>  |
| <b>Syntax</b>        | <code>void Server_Done();</code>  |
| <b>Arguments</b>     | None  |
| <b>Return Value</b>  | None  |
| <b>Description</b>   | <p>This function notifies the SDB server that the application has finished servicing the command. Once the server receives a command from the host client, it does not accept any more commands until this function is called. Also, after this function is called, the 'C80 application cannot call <code>Server_GetCmnd()</code> until the server receives a new command.</p> |
| <b>Example</b>       | <code>Server_Done( );</code>  |

|                      |   |
|----------------------|---|
| <b>Function Name</b> | <b>Server_GetCmnd</b>   |
| <b>Syntax</b>        | <code>void Server_GetCmnd(ULONG *Cmnd);</code>  |
| <b>Arguments</b>     | <code>ULONG *Cmnd</code> Pointer to a series of 32-bit words whose count is the value of SERVER_CMND_SIZE   |
| <b>Return Value</b>  | None  |
| <b>Description</b>   | This function copies the command read by the 'C80 command server into the command pointed to by Cmnd. The Cmnd argument must point to a series of 32-bit words of preallocated memory whose count is the value of SERVER_CMND_SIZE. The server command is copied only if the server has one waiting and the application has not yet called Server_Done() for that command. Otherwise, *Cmnd is filled with zeros. |
| <b>Example</b>       | <pre>ULONG Cmnd[SERVER_CMND_SIZE]; <b>Server_GetCmnd(Cmnd);</b> Server_Sync(); /* ... complete command ... */</pre>   |

|                      |   |               |   |
|----------------------|---|---------------|---|
| <b>Function Name</b> | <b>Server_Init</b>  |               |   |
| <b>Syntax</b>        | BOOL Server_Init();   |               |   |
| <b>Arguments</b>     | None  |               |   |
| <b>Return Value</b>  | BOOL  | TRUE<br>FALSE | Initialization succeeded<br>Initialization failed |
| <b>Description</b>   | This function initializes the server API. It returns TRUE if the initialization succeeded.  |               |   |
|                      | <div><b>Notes:</b><ol style="list-style-type: none"><li>1) You must call Server_Init() <i>before</i> calling any other server API function.</li><li>2) You must call this function only once in an application.</li></ol></div> |               |   |
| <b>Example</b>       | <pre>BOOL ok;<br/><br/>ok = Server_Init();<br/><br/>if (ok) {<br/>    /* ... do some processing ... */<br/>}</pre>  |               |   |

|                      |   |
|----------------------|---|
| <b>Function Name</b> | <b>Server_Install</b>   |
| <b>Syntax</b>        | <code>void Server_Install();</code>   |
| <b>Arguments</b>     | None  |
| <b>Return Value</b>  | None  |
| <b>Description</b>   | This function installs the server ISR. <code>Server_Install()</code> must be called before the host (client) can send down any commands. Calling <code>Server_UnInstall()</code> reverses the actions of this function. |
| <b>Example</b>       | <pre>Server_Init();<br/><b>Server_Install();</b></pre>  |

|                      |   |
|----------------------|---|
| <b>Function Name</b> | <b>Server_InstallSema</b>   |
| <b>Syntax</b>        | long Server_InstallSema(long <i>SemaId</i> );   |
| <b>Arguments</b>     | long <i>SemaId</i> ID of an opened semaphore  |
| <b>Return Value</b>  | long              Old semaphore value   |
| <b>Description</b>   | This function installs a semaphore into the server driver. Whenever the SDB receives a new command from the host (client), it signals this semaphore. The application must open the semaphore before installing it. |
| <b>Example</b>       | <pre>long ServerSemaId;  ServerSemaId = TaskOpenSema(-1,0); <b>Server_InstallSema(ServerSemaId);</b>  . . .  TaskWaitSema(ServerSemaId);</pre>  |

|                      |   |
|----------------------|---|
| <b>Function Name</b> | <b>Server_ReadDataFifo</b>  |
| <b>Syntax</b>        | <code>long Server_ReadDataFifo(ULONG *Dst, USHORT Length);</code>   |
| <b>Arguments</b>     | <code>ULONG *Dst</code> Pointer to source of transfer<br><code>USHORT Length</code> Number of 32-bit words to read ( $0 < Length \leq 8192$ )   |
| <b>Return Value</b>  | None  |
| <b>Description</b>   | <p>This function transfers a block of data from the PCI interface FIFO to SDB memory by setting up a packet transfer and then issuing it. The function waits on a packet complete semaphore, so it cannot be called from the default task.</p> <div><b>Note:</b><br/>The length of the packet transfer is limited to 8192 32-bit words.</div> |
| <b>Example</b>       | <pre>ULONG MyBuff[64];<br/><br/>Server_ReadDataFifo(MyBuff, 64);</pre>  |



|                      |  |
|----------------------|--|
| <b>Function Name</b> | <b>Server_Sync</b>   |
| <b>Syntax</b>        | <code>void Server_Sync();</code>   |
| <b>Arguments</b>     | None   |
| <b>Return Value</b>  | None   |
| <b>Description</b>   | When you call this function, it waits until the host executes <code>Client_Sync()</code> before returning. This synchronizes the 'C80 server with the host client. |
| <b>Example</b>       | <code>Server_Sync ( ) ;</code>   |

|                      |  |
|----------------------|--|
| <b>Function Name</b> | <b>Server_UnInstall</b>  |
| <b>Syntax</b>        | <code>void Server_UnInstall();</code>  |
| <b>Arguments</b>     | None   |
| <b>Return Value</b>  | None   |
| <b>Description</b>   | <p>This function performs the following actions:</p> <ul style="list-style-type: none"><li><input type="checkbox"/> Uninstalls the server that was installed by calling <code>Server_Install()</code></li><li><input type="checkbox"/> Disables the server interrupt</li></ul> |
| <b>Example</b>       | <pre>Server_UnInstall();</pre>   |

|                      |  |
|----------------------|--|
| <b>Function Name</b> | <b>Server_WriteDataFifo</b>  |
| <b>Syntax</b>        | <code>void Server_WriteDataFifo(ULONG *Src, USHORT Length);</code>   |
| <b>Arguments</b>     | <code>ULONG *Src</code> Pointer to source of transfer<br><code>USHORT Length</code> Number of 32-bit words to read ( $0 < Length \leq 8192$ )  |
| <b>Return Value</b>  | None   |
| <b>Description</b>   | <p>This function transfers a block of data from SDB memory to the PCI interface by setting up a packet transfer and then issuing it. The function waits on a packet complete semaphore, so it cannot be called from the default task.</p> <div><b>Note:</b><br/>The length of the packet transfer is limited to 8192 32-bit words.</div> |
| <b>Example</b>       | <pre>ULONG MyBuff[64];<br/><br/>/* write the contents of the buffer to the FIFO */<br/>Server_WriteDataFifo(MyBuff, 64);</pre>   |

## Example Code

This appendix lists several examples of code that illustrate the effective use of the API libraries that come with the SDB. Some of these examples may be good starting points for developing your own applications.

The complete coding examples illustrate how to use the driver APIs. Each example includes the following files:

- ☐ C source file for the example
- ☐ Linker command file
- ☐ Batch file for building the project

The project files are located on the *TMS320C8x SDB System Software* CD ROM under the `samples` directory. The projects on the CD have already been built into a `.out` file and are ready to execute. If you want to rebuild the projects, the TMS320C8x code generations tools must be installed on your system.

| Topic  | Page        |
|--|-------------|
| <b>A.1 Video Capture-Process-Display Example .....</b> | <b>A-2</b>  |
| <b>A.2 Audio DMA Capture Example .....</b>             | <b>A-8</b>  |
| <b>A.3 Audio DMA Playback Example .....</b>            | <b>A-12</b> |
| <b>A.4 Audio Block Capture/Playback Example .....</b>  | <b>A-17</b> |
| <b>A.5 Audio Programmed I/O Example .....</b>          | <b>A-21</b> |
| <b>A.6 Video Capture Scaling Example .....</b>         | <b>A-24</b> |
| <b>A.7 Video Display Test Example .....</b>            | <b>A-30</b> |

## A.1 Video Capture-Process-Display Example

This example provides a skeleton application that performs double-buffered video capture and double-buffered video display. You can start with this example and add to it by providing processing on the captured video.

### Example A–1. video

#### (a) video.c

```
/* *****\
 *      Copyright (C) 1995-1996 Texas Instruments Incorporated.
 *      All Rights Reserved
 *-----
 *
 * video.c  -- TMS320C8x MP 'C' Source Code
 *
 *      This is the C file for the video example program.
 *
\*****/
#include <sdbdrvs.h>

/* *****\
 *      Data Types, Global Variables, Constants
\*****/

/* executive variables */
long VideoTaskId;
long CaptureSemaId;
long DisplaySemaId;
long PtSemaId;

/* packet transfer table (PT is defined in sdbdrvs.h) */
PT *PtTable;

/* metrics variables */
CAPTURE_MET CM;
DISPLAY_MET DM;

/* *****\
 *      Function Prototypes
\*****/
void VideoTask(void *p);
void ProcessVideo(ULONG Buff);
void InitVideo(USHORT sx, USHORT sy, BYTE scale);
void InitPtTable();
```

```

/*****\
*
*                               Functions
\*****/
void main() {

    /* these register settings are mandatory */
    REFCNTL = 0xFFFF0100;
    PTMIN   = 0x00000100;
    PTMAX   = 0x00010000;

    /* initialize the executive */
    TaskInitTasking();
    PtReqInit();

    /* initialize the drivers */
    Capture_Init();
    Display_Init();

    /* create the task and semaphores */
    VideoTaskId = TaskCreate(-1, VideoTask, (void *)NULL, 20, 1024);
    CaptureSemaId = TaskOpenSema(-1,0);
    DisplaySemaId = TaskOpenSema(-1,0);

    /* install driver semaphores */
    Capture_InstallSema(CaptureSemaId);
    Display_InstallSema(DisplaySemaId);

    /* set up packet transfer table and semaphore */
    PtSemaId = TaskOpenSema(-1,0);
    PtTable = (PT *)PtReqAlloc();

    /* start the video task */
    TaskResume(VideoTaskId);

    while (1);
}

/*****\
void VideoTask(void *p) {

    int  SemaCt = 0;
    ULONG CaptureBuff;
    ULONG DisplayBuff;

    /* initialize the video settings */
    if (0) InitVideo(1024, 768, CAPTURE_640x480);
    if (1) InitVideo( 640, 480, CAPTURE_640x480);
    if (0) InitVideo(1024, 768, CAPTURE_512x512);
    if (0) InitVideo( 640, 480, CAPTURE_CIF);
    if (0) InitVideo( 640, 480, CAPTURE_CIFK);
    if (0) InitVideo( 640, 480, CAPTURE_QCIF);
    if (0) InitVideo( 640, 480, CAPTURE_SQCIF);
    if (0) InitVideo(1024, 768, CAPTURE_SQCIF);
}

```

```
/* turn on the video */
Display_Enable();
Capture_Enable();

/* video capture-process-display loop */
while (1) {

    /* wait for new captured frame */
    TaskWaitSema(CaptureSemaId);

    /* selectively process frame */
    if (SemaCt++ > 0) {
        SemaCt = 0;

        /* get buffer of most recently captured frame */
        CaptureBuff = Capture_GetBuffer();

        /* tell display driver to toggle buffers next display frame event */
        Display_ToggleBuffers();

        /* do some processing on captured buffer */
        if (0) ProcessVideo(CaptureBuff);

        /* wait until display driver has toggled display buffers */
        TaskWaitSema(DisplaySemaId);

        /* get the in-active display buffer */
        DisplayBuff = Display_GetBuffer(DISPLAY_INACTIVE);

        /* transfer the processed capture buffer to the */
        /* in-active display buffer */
        PtTable->SrcStart = CaptureBuff;
        PtTable->DstStart = DisplayBuff;
        PtReqIssue((void *)PtTable,PtSemaId);
        TaskWaitSema(PtSemaId);

        /* unlock the capture buffer obtained from Capture_GetBuffer() */
        Capture_FreeBuffer();
    }
}

/*-----*/
void ProcessVideo(ULONG Buff) {

    USHORT x,y;
    ULONG  addr;

    /* this processing is not practical but has a good visual effect */
}
```

```

/* for each 16-bit pixel in the captured buffer */
for (x=0; x<CM.Rh; x++) {
    for (y=0; y<CM.Rv; y++) {
        addr = Buff + CM.Bpp/8*x + CM.Pitch*y;
        /* invert the pixel */
        NOCACHE_USHORT(*(volatile USHORT*)addr) =
            (0xFFFF - NOCACHE_USHORT(*(volatile USHORT*)addr));
    }
}
}
/*-----*/
void InitVideo(USHORT sx, USHORT sy, BYTE scale) {
    USHORT x,y,dx,dy;

    /* set up the video capture */
    Capture_Install(CAPTURE_NTSC,CAPTURE_RGB555,scale);
    Capture_FillBuffs(0x00000000);
    Capture_GetMetrics(&CM);

    /* calculate the display window */
    dx = CM.Rh;
    dy = CM.Rv;
    x  = (sx-dx)/2;
    y  = (sy-dy)/2;

    /* set up the display */
    Display_SetMode(sx,sy,60,DISPLAY_T555,DISPLAY_VIDEO);
    Display_SetWindow(x,y,dx,dy);
    Display_SetPitch(CM.Pitch);
    Display_GetMetrics(&DM);

    /* initialize the packet transfer table */
    InitPtTable();
}

/*-----*/
void InitPtTable() {

    /* Setup packet transfer table used to transfer the capture buffer to
    /* the display buffer. The two metrics variables CM and DM must already
    /* be set before calling this function. */
    PtTable->Next      = (ULONG)PtTable;
    PtTable->Options    = 0x80000000;
    PtTable->SrcStart   = 0x00000000;
    PtTable->DstStart   = 0x00000000;
    PtTable->SrcBCnt    = CM.Rv-1;
    PtTable->SrcACnt    = CM.Rh*CM.Bpp/8;
    PtTable->DstBCnt    = CM.Rv-1;
    PtTable->DstACnt    = CM.Rh*CM.Bpp/8;
    PtTable->SrcCCnt    = 0;
    PtTable->DstCCnt    = 0;
    PtTable->SrcBPitch  = CM.Pitch;
    PtTable->DstBPitch  = DM.Pitch;
}

```



```
PtTable->SrcCPitch = 0;
PtTable->DstCPitch = 0;
PtTable->Trans0    = 0;
PtTable->Trans1    = 0;
PtTable->Junk1     = 0;
PtTable->Junk2     = 0;
}

/*****\
* End of 'video.c'
*****/

(b) video.lnk
/*****\
*      Copyright (C) 1995-1996 Texas Instruments Incorporated.
*                  All Rights Reserved
*-----
*
* video.lnk  -- TMS320C8x MP Linker Command File
*
*****/

-c
-x
-heap 0x00400000
-stack 0x00010000
-l mp_rts.lib
-l mp_task.lib
-l mp_int.lib
-l mp_ptreq.lib
-l sdbdrvs.lib

MEMORY
{
    PROGMEM : origin=0x80000000 length=0x00800000
}

SECTIONS
{
    .text    :> PROGMEM
    .ptext   :> PROGMEM
    .cinit   :> PROGMEM
    .const   :> PROGMEM
    .switch  :> PROGMEM
    .data    :> PROGMEM
    .bss     :> PROGMEM
    .system  :> PROGMEM
}

/*****\
* End of 'video.lnk'
*****/
```

(c) video.bat

```
@rem #*****\
@rem #      Copyright (C) 1995-1996 Texas Instruments Incorporated.
@rem #                      All Rights Reserved
@rem #-----
@rem #
@rem #  video.bat  -- batch file to build project
@rem #
@rem #*****/

@mpcl -gq video.c

@mvplnk -m video.map -o video.out video.obj video.lnk

@rem #*****\
@rem # End of 'video.bat'
@rem #*****/
```

## A.2 Audio DMA Capture Example

This example shows how to use buffered DMA audio capture in real time. The only processing done on the audio is displaying each sample on the display in an oscilloscope fashion. When you run this example in the debugger, do not halt the debugger. The audio FIFO generates an interrupt when it is almost full. The ISR then issues a packet transfer to read the data out of the FIFO. If you halt the debugger, the ISR never executes, and the FIFO is never serviced. When this happens, the FIFO becomes full and overflows. No more events are triggered and, therefore, audio stops.

### Example A–2. *audcapt*

#### (a) *audcapt.c*

```
/* *****\
 *      Copyright (C) 1995–1996 Texas Instruments Incorporated.
 *      All Rights Reserved
 *-----
 *
 *  audcapt.c  -- TMS320C8x MP 'C' Source Code
 *
 *  This is the C file for the audcapt example.
 *
 * *****/
#include <sdbdrvs.h>

/* *****\
 *      Data Types, Global Variables, Constants
 * *****/
long AudioTaskId;
long AudioSemaId;

AUDIO_MET AM;
DISPLAY_MET DM;

#define BG_COLOR 0x0000
#define FG_COLOR 0x03E0

/* *****\
 *      Function Prototypes
 * *****/
void AudioTask(void *p);
void ProcessAudio(AUDIO_PTR *Cptr);

/* *****\
 *      Functions
 * *****/
void main() {

    /* these register settings are mandatory */
    REFCNTL = 0xFFFF0100;
```

```

PTMIN    = 0x00000100;
PTMAX    = 0x00010000;

/* initialize executive */
TaskInitTasking();
PtReqInit();

/* initialize drivers */
Audio_Init();
Display_Init();

/* set up display */
Display_SetMode(1024,768,60,DISPLAY_T555,DISPLAY_VIDEO);
Display_FillBufs(BG_COLOR);
Display_GetMetrics(&DM);
Display_Enable();

/* executive stuff */
AudioSemaId = TaskOpenSema(-1,0);
AudioTaskId = TaskCreate(-1, AudioTask, (void *)NULL, 20, 1024);
Audio_InstallSema(AudioSemaId);

/* start the audio task */
TaskResume(AudioTaskId);

while (1);
}

/*-----*/
void AudioTask(void *p) {

    AUDIO_PTR Cptr;

    /* set up the audio */
    Audio_ProgramInputs(AUDIO_LINE,AUDIO_LINE,15,15);
    Audio_Install(AUDIO_CAPTURE,AUDIO_PCM16,AUDIO_STEREO,48.0,8,128,4);
    Audio_FillBufs(0x00000000);
    Audio_GetMetrics(&AM);
    Audio_Enable();

    /* audio capture loop */
    while (1) {
        TaskWaitSema(AudioSemaId);
        Audio_GetCaptureBufs(&Cptr);
        ProcessAudio(&Cptr);
    }
}

/*-----*/
void ProcessAudio(AUDIO_PTR *Cptr) {

    USHORT x,y,n;
    static BOOL init = FALSE;

```

```

static short L0[2048];
static short R0[2048];

n = AM.BlockSz * AM.BlockCt;

if (init) {
    for (x=0; x<n; x++) {
        Display_SetPixel(x,L0[x]+768/3,BG_COLOR,DISPLAY_ACTIVE);
        L0[x] = ((short*)(Cptr->L))[x]>>8;
        Display_SetPixel(x,768/3,0x3C00,DISPLAY_ACTIVE);
        Display_SetPixel(x,L0[x]+768/3,FG_COLOR,DISPLAY_ACTIVE);
        if (AM.Stereo) {
            Display_SetPixel(x,R0[x]+2*768/3,BG_COLOR,DISPLAY_ACTIVE);
            R0[x] = ((short*)(Cptr->R))[x]>>8;
            Display_SetPixel(x,2*768/3,0x3C00,DISPLAY_ACTIVE);
            Display_SetPixel(x,R0[x]+2*768/3,FG_COLOR,DISPLAY_ACTIVE);
        }
    }
}
else {
    init=TRUE;
    for (x=0; x<n; x++) {
        L0[x] = R0[x] = 0;
    }
}
}

/*****\
* End of 'audcapt.c'
\*****/

(b) audcapt.lnk
/*****\
*      Copyright (C) 1995-1996 Texas Instruments Incorporated.
*                  All Rights Reserved
*-----
*
* audcapt.lnk -- TMS320C8x MP Linker Command File
*
\*****/

-c
-x
-heap 0x00100000
-stack 0x00010000
-l mp_rts.lib
-l mp_task.lib
-l mp_int.lib
-l mp_ptreq.lib
-l sdbdrvs.lib

```

```

MEMORY
{
    PROGMEM : origin=0x80000000 length=0x00800000
}

SECTIONS
{
    .text    :> PROGME
    .ptext   :> PROGME
    .cinit   :> PROGME
    .const   :> PROGME
    .switch  :> PROGME
    .data    :> PROGME
    .bss     :> PROGME
    .system  :> PROGME
}
/*****\
* End of 'audcapt.lnk'
\*****/

(c) audcapt.bat
@rem #*****\
@rem #      Copyright (C) 1995-1996 Texas Instruments Incorporated.
@rem #                               All Rights Reserved
@rem #-----
@rem #
@rem #  audcapt.bat  -- batch file to build project
@rem #
@rem #*****/

mpcl -gq audcapt.c

mvplnk -m audcapt.map -o audcapt.out audcapt.obj audcapt.lnk

@rem #*****\
@rem # End of 'audcapt.bat'
@rem #*****/

```

### A.3 Audio DMA Playback Example

This example shows how to use buffered DMA audio playback in real time. The audio source for playback is simulated by generating a sine wave. The only processing done on the audio is displaying each sample on the display in an oscilloscope fashion. When you run this example in the debugger, do not halt the debugger. The audio FIFO generates an interrupt when it is almost empty. The ISR then issues a packet transfer to write the audio data out to the FIFO. If you halt the debugger, the ISR never executes, and the FIFO is never serviced. When this happens, the FIFO empties and underflows. No more events are triggered and, therefore, audio stops.

#### Example A–3. *audplay*

##### (a) *audplay.c*

```
/* *****\
 *      Copyright (C) 1995–1996 Texas Instruments Incorporated.
 *      All Rights Reserved
 *-----
 *
 *  audplay.c  -- TMS320C8x MP 'C' Source Code
 *
 *  This is the C file for the audplay module.
 *
\*****/
#include <sdbdrvs.h>

/* *****\
 *      Data Types, Global Variables, Constants
\*****/
long AudioTaskId;
long AudioSemaId;

AUDIO_MET AM;
DISPLAY_MET DM;

#define BG_COLOR 0x0000
#define FG_COLOR 0x03E0

/* *****\
 *      Function Prototypes
\*****/
void AudioTask(void *p);
void ProcessAudio(AUDIO_PTR *Pptr);
```

```

/*****\
*
*                               Functions
\*****/
void main() {

    /* these register settings are mandatory */
    REFCNTL = 0xFFFF0100;
    PTMIN   = 0x00000100;
    PTMAX   = 0x00010000;

    /* initialize executive */
    TaskInitTasking();
    PtReqInit();

    /* initialize drivers */
    Audio_Init();
    Display_Init();

    /* set up display */
    Display_SetMode(1024,768,60,DISPLAY_T555,DISPLAY_VIDEO);
    Display_FillBufs(BG_COLOR);
    Display_GetMetrics(&DM);
    Display_Enable();

    /* executive stuff */
    AudioSemaId = TaskOpenSema(-1,0);
    AudioTaskId = TaskCreate(-1, AudioTask, (void *)NULL, 20, 1024);
    Audio_InstallSema(AudioSemaId);

    /* start the audio task */
    TaskResume(AudioTaskId);

    while (1);
}

/*****\
void AudioTask(void *p) {

    AUDIO_PTR Pptr;

    /* set up the audio */
    Audio_ProgramInputs(AUDIO_LINE,AUDIO_LINE,15,15);
    Audio_Install(AUDIO_PLAYBACK,AUDIO_PCM16,AUDIO_STEREO,48.0,8,128,4);
    Audio_FillBufs(0x00000000);
    Audio_GetMetrics(&AM);
    Audio_Enable();

    /* audio capture loop */
    while (1) {
        TaskWaitSema(AudioSemaId);
    }
}

```



```

    Audio_GetPlaybackBufs(&Pptr);
    ProcessAudio(&Pptr);
}

/*-----*/
void ProcessAudio(AUDIO_PTR *Pptr) {

    USHORT x,y,n;
    static BOOL init = FALSE;
    static short L0[2048];
    static short R0[2048];
    static float theta = 0.0;
    static float pi;
    float z;

    pi = 4.0*atan(1.0);
    n = AM.BlockCt * AM.BlockSz;

    /* erase old plot */
    if (init) {
        for (x=0; x<n; x++) {
            Display_SetPixel(x,L0[x]+768/3,BG_COLOR,DISPLAY_ACTIVE);
            if (AM.Stereo)
                Display_SetPixel(x,R0[x]+2*768/3,BG_COLOR,DISPLAY_ACTIVE);
        }
    }

    init=TRUE;

    /*theta = 0.0;*/
    /* draw new plot */
    for (x=0; x<n; x++) {
        z = 16384*sin(theta);
        L0[x] = (short)z>>8;
        NOCACHE_SHORT(*(volatile short *)((short*)(Pptr->L)+x)) = (short)z;
        Display_SetPixel(x,768/3,0x3C00,DISPLAY_ACTIVE);
        Display_SetPixel(x,L0[x]+768/3,FG_COLOR,DISPLAY_ACTIVE);
        if (AM.Stereo) {
            z = 16384*sin(4*theta+pi);
            R0[x] = (short)z>>8;
            NOCACHE_SHORT(*(volatile short *)((short*)(Pptr->R)+x)) = (short)z;
            Display_SetPixel(x,2*768/3,0x3C00,DISPLAY_ACTIVE);
            Display_SetPixel(x,R0[x]+2*768/3,FG_COLOR,DISPLAY_ACTIVE);
        }
        theta = theta + 4.0*2.0*pi/n;
    }
}

/*****\
* End of 'audplay.c'
\*****/

```

*(b) audplay.lnk*

```

/*****\
*      Copyright (C) 1995-1996 Texas Instruments Incorporated.
*                      All Rights Reserved
*-----
*
*  audplay.lnk  --  TMS320C8x MP Linker Command File
*
\*****/

-c
-x
-heap 0x00100000
-stack 0x00010000
-l mp_rts.lib
-l mp_task.lib
-l mp_int.lib
-l mp_ptreq.lib
-l sdbdrvs.lib

MEMORY
{
    PROGMEM : origin=0x80000000 length=0x00800000
}

SECTIONS
{
    .text    :> PROGMEM
    .ptext   :> PROGMEM
    .cinit   :> PROGMEM
    .const   :> PROGMEM
    .switch  :> PROGMEM
    .data    :> PROGMEM
    .bss     :> PROGMEM
    .system  :> PROGMEM
}

/*****\
* End of 'audplay.lnk'
\*****/

```

*(c) audplay.bat*

```

@rem # *****\
@rem #      Copyright (C) 1995-1996 Texas Instruments Incorporated.
@rem #                      All Rights Reserved
@rem #-----
@rem #
@rem #  audplay.bat  --  batch file to build project
@rem #
@rem # *****/

mpc1 -gq audplay.c

```

```
mvplnk -m audplay.map -o audplay.out audplay.obj audplay.lnk
```

```
@rem #*****\
@rem # End of 'audplay.bat'
@rem #*****/
```

## A.4 Audio Block Capture/Playback Example

This example shows how to use the audio block processing functions, `Audio_CaptureToMemory()` and `Audio_PlaybackFromMemory()`. The program captures 10 seconds of audio, and then plays it back. You can insert processing between the capture and playback.

### Example A-4. *audtest*

#### (a) *audtest.c*

```

/*****
 *      Copyright (C) 1995-1996 Texas Instruments Incorporated.
 *
 *      All Rights Reserved
 *
 *-----
 *
 *  audtest.c  -- TMS320C8x MP 'C' Source Code
 *
 *  This is the C file for the audtest module.
 *
 *****/
#include <sdbdrvs.h>
#include <stdlib.h>

/*****
 *
 *      Data Types, Global Variables, Constants
 *****/

/* executive stuff */
long AudioTaskId;
long AudioSemaId;

/* metrics variables */
AUDIO_MET AM;

/*****
 *
 *      Function Prototypes
 *****/
void AudioTask(void *p);
void ProcessAudio(AUDIO_PTR *Cptr, ULONG NumBufs);

/*****
 *
 *      Functions
 *****/
void main() {

    /* these register settings are mandatory */
    REFCNTL = 0xFFFF0100;
    PTMIN   = 0x00000100;
    PTMAX   = 0x00010000;

    /* initialize executive */
    TaskInitTasking();

```

```
PtReqInit();

/* initialize drivers */
Audio_Init();

/* executive stuff */
AudioSemaId = TaskOpenSema(-1,0);
AudioTaskId = TaskCreate(-1, AudioTask, (void *)NULL, 20, 1024);
Audio_InstallSema(AudioSemaId);

/* start the audio task */
TaskResume(AudioTaskId);

while (1);
}

/*-----*/
void AudioTask(void *p) {

    AUDIO_PTR Cptr;
    BYTE  Format   = AUDIO_PCM16;
    BOOL  Stereo   = AUDIO_STEREO;
    float Fs       = 48.0;
    BYTE  BlockCt  = 20;
    BYTE  BlockSz  = 50;
    BYTE  BuffCt   = 8;
    ULONG NumBufs  = Fs*10; /* 10 seconds worth (watch out for heap size) */

    /* dummy install to set audio metrics */
    Audio_Install(AUDIO_CAPTURE,Format,Stereo,Fs,BlockCt,BlockSz,BuffCt);
    Audio_GetMetrics(&AM);
    Audio_UnInstall();

    /* allocate some DRAM storage for captured audio */
    Cptr.L = (void*)memalign(64,AM.ByteSz*NumBufs);
    Cptr.R = (void*)memalign(64,AM.ByteSz*NumBufs);

    /* set the input gain */
    Audio_ProgramInputs(AUDIO_LINE,AUDIO_LINE,9,9);

    while (1) {
        /* capture audio into memory */
        Audio_Install(AUDIO_CAPTURE,Format,Stereo,Fs,BlockCt,BlockSz,BuffCt);
        Audio_CaptureToMemory(&Cptr, NumBufs);
        Audio_UnInstall();

        /* process the captured audio */
        ProcessAudio(&Cptr,NumBufs);

        /* playback the processed audio */
        Audio_Install(AUDIO_PLAYBACK,Format,Stereo,Fs,BlockCt,BlockSz,BuffCt);
```

```

    Audio_PlaybackFromMemory(&Cptr, NumBufs);
    Audio_UnInstall();
}

/* if loop exited, free up the allocated memory */
free((void*)Cptr.L);
free((void*)Cptr.R);
}

/*-----*/
void ProcessAudio(AUDIO_PTR *Cptr, ULONG NumBufs) {

    ULONG i;
    short *L,*R;

    L = (short*)(Cptr->L);
    R = (short*)(Cptr->R);

    /* do some dummy processing */
    /* watch out for data cache coherency */
    for (i=0; i<AM.BuffSz*NumBufs; i++) {
        L[i] = -L[i];
        R[i] = R[i]/2;
    }
}

/*****\
* End of 'audtest.c'
*****/

```

**(b) audtest.lnk**

```

/*****\
*      Copyright (C) 1995-1996 Texas Instruments Incorporated.
*      All Rights Reserved
*-----*
*      audtest.lnk  -- TMS320C8x MP Linker Command File
*
*****/

-c
-x
-heap 0x00400000
-stack 0x00010000
-l mp_ptreq.lib
-l mp_rts.lib
-l mp_task.lib
-l mp_int.lib
-l sdbdrvs.lib

MEMORY
{

```

```
    PROGMEM : origin=0x80000000 length=0x00800000
}

SECTIONS
{
    .text    :> PROGMEM
    .stack   :> PROGMEM
    .ptext   :> PROGMEM
    .cinit   :> PROGMEM
    .const   :> PROGMEM
    .switch  :> PROGMEM
    .data    :> PROGMEM
    .bss     :> PROGMEM
    .sysmem  :> PROGMEM
}

/*****\
* End of 'audtest.lnk'
*****/

(c) audtest.bat
@rem #*****\
@rem #      Copyright (C) 1995-1996 Texas Instruments Incorporated.
@rem #                      All Rights Reserved
@rem #-----
@rem #
@rem #  audtest.bat  -- batch file to build project
@rem #
@rem #*****/

mpc1 -qg audtest.c

mvplnk -m audtest.map -o audtest.out audtest.obj audtest.lnk

@rem #*****\
@rem # End of 'audtest.bat'
@rem #*****/
```

## A.5 Audio Programmed I/O Example

This example shows how to use the PIO (programmed I/O) mode of the audio codec to perform full-duplex audio. Two methods are used:

- ☐ The program creates its own PIO loop.
- ☐ The program calls the built-in driver function `Audio_PioTest()`.

This example tests many different audio formats.

### Example A-5. *piotest*

#### (a) *piotest.c*

```

/*****
 *      Copyright (C) 1995-1996 Texas Instruments Incorporated.
 *      All Rights Reserved
 *-----
 *
 *  piotest.c  -- TMS320C8x MP 'C' Source Code
 *
 *      This is the C file for the piotest example.
 *
 \*****/
#include <sdbdrvs.h>

/*****
 *
 *      Functions
 \*****/
void main() {

    short L,R;
    ULONG ct = 48000*10;
    ULONG x;

    /* these register settings are mandatory */
    REFCNTL = 0xFFFF0100;
    PTMIN   = 0x00000100;
    PTMAX   = 0x00010000;

    /* initialize the executive */
    TaskInitTasking();
    PtReqInit();

    /* initialize the audio driver */
    Audio_Init();

    /* setup PIO mode */
    Audio_Install(AUDIO_PIO, AUDIO_PCM16, AUDIO_STEREO, 48.0, 0,0,0);
    Audio_ProgramInputs(AUDIO_LINE, AUDIO_LINE, 9.0, 9.0);
    Audio_Enable();

```



```

/* do loopback for ct samples */
for (x=0; x<ct; x++) {
    Audio_PioIn(&L,&R);
    Audio_PioOut(&L,&R);
}

/* now use the driver built in PIO test routine */
Audio_PioTest(AUDIO_PCM16, AUDIO_STEREO, 48.0, 48000*60);

Audio_PioTest(AUDIO_PCM16, AUDIO_STEREO, 48.0, 48000*5);
Audio_PioTest(AUDIO_PCM16, AUDIO_MONO, 48.0, 48000*5);
Audio_PioTest(AUDIO_PCM16, AUDIO_STEREO, 8.0, 8000*5);
Audio_PioTest(AUDIO_PCM16, AUDIO_MONO, 8.0, 8000*5);

Audio_PioTest(AUDIO_PCM8, AUDIO_STEREO, 48.0, 48000*5);
Audio_PioTest(AUDIO_PCM8, AUDIO_MONO, 48.0, 48000*5);
Audio_PioTest(AUDIO_PCM8, AUDIO_STEREO, 8.0, 8000*5);
Audio_PioTest(AUDIO_PCM8, AUDIO_MONO, 8.0, 8000*5);

Audio_PioTest(AUDIO_ALAW8, AUDIO_STEREO, 48.0, 48000*5);
Audio_PioTest(AUDIO_ALAW8, AUDIO_MONO, 48.0, 48000*5);
Audio_PioTest(AUDIO_ALAW8, AUDIO_STEREO, 8.0, 8000*5);
Audio_PioTest(AUDIO_ALAW8, AUDIO_MONO, 8.0, 8000*5);

Audio_PioTest(AUDIO_ULAW8, AUDIO_STEREO, 48.0, 48000*5);
Audio_PioTest(AUDIO_ULAW8, AUDIO_MONO, 48.0, 48000*5);
Audio_PioTest(AUDIO_ULAW8, AUDIO_STEREO, 8.0, 8000*5);
Audio_PioTest(AUDIO_ULAW8, AUDIO_MONO, 8.0, 8000*5);

Audio_PioTest(AUDIO_PCM16, AUDIO_STEREO, 48.0, -1);

while (1);
}

/*****\
* End of 'piotest.c'
*****/

(b) piotest.lnk
/*****\
* Copyright (C) 1995-1996 Texas Instruments Incorporated.
* All Rights Reserved
*-----
*
* piotest.lnk -- TMS320C8x MP Linker Command File
*
*****/

-c
-x
-heap 0x00100000
-stack 0x00010000

```

```
-l mp_rts.lib
-l mp_task.lib
-l mp_int.lib
-l mp_ptreq.lib
-l sdbdrvs.lib
```

```
MEMORY
{
    PROGMEM : origin=0x80000000 length=0x00800000
}
```

```
SECTIONS
{
    .text    :> PROGMEM
    .ptext   :> PROGMEM
    .cinit   :> PROGMEM
    .const   :> PROGMEM
    .switch  :> PROGMEM
    .data    :> PROGMEM
    .bss     :> PROGMEM
    .system  :> PROGMEM
}
```

```
/*****\
* End of 'piotest.lnk'
*****/
```

### (c) *piotest.bat*

```
@rem # *****\
@rem #      Copyright (C) 1995-1996 Texas Instruments Incorporated.
@rem #                               All Rights Reserved
@rem #-----
@rem #
@rem #   piotest.bat  -- batch file to build project
@rem #
@rem # *****/

@mpc1 -gq piotest.c

@mvplnk -m piotest.map -o piotest.out piotest.obj piotest.lnk

@rem # *****\
@rem # End of 'piotest.bat'
@rem # *****/
```

## A.6 Video Capture Scaling Example

This example shows how to set up video capture and then change the scaling. It displays each scaling setting in its own box on the display. For additional information about video capture scaling, see subsection 5.1.1, *Supported Scaling Resolutions*, on page 5-4.

### Example A–6. *captttest*

#### (a) *captttest.c*

```
/* *****\
 *      Copyright (C) 1995–1996 Texas Instruments Incorporated.
 *      All Rights Reserved
 *-----
 *
 *  captttest.c  -- TMS320C8x MP 'C' Source Code
 *
 *  This is the C file for the captttest example.
 *
\*****/
#include <sdbdrvs.h>

/* *****\
 *      Data Types, Global Variables, Constants
\*****/

/* executive variables */
long VideoTaskId;
long CaptureSemaId;
long DisplaySemaId;
long PtSemaId;

/* packet transfer table (PT is defined in sdbdrvs.h) */
PT *PtTable;

/* metrics variables */
CAPTURE_MET CM;
DISPLAY_MET DM;

/* *****\
 *      Function Prototypes
\*****/
void VideoTask(void *p);
void DoCapture(ULONG frames, BYTE scale);
void InitPtTable();
void DrawScreen();
void DrawBox(USHORT dx, USHORT dy, ULONG color);

/* *****\
 *      Functions
\*****/
void main() {
```

```

/* these register settings are mandatory */
REFCNTL = 0xFFFF0100;
PTMIN   = 0x00000100;
PTMAX   = 0x00010000;

/* initialize the executive */
TaskInitTasking();
PtReqInit();

/* initialize the drivers */
Capture_Init();
Display_Init();

/* create the task and semaphores */
VideoTaskId = TaskCreate(-1, VideoTask, (void *)NULL, 20, 1024);
CaptureSemaId = TaskOpenSema(-1,0);
DisplaySemaId = TaskOpenSema(-1,0);

/* install driver semaphores */
Capture_InstallSema(CaptureSemaId);
Display_InstallSema(DisplaySemaId);

/* set up packet transfer table and semaphore */
PtSemaId = TaskOpenSema(-1,0);
PtTable = (PT *)PtReqAlloc();

/* start the video task */
TaskResume(VideoTaskId);

while (1);
}

/*-----*/
void VideoTask(void *p) {

    /* number of seconds run each capture setting */
    USHORT frames = 30*5; /* 5 seconds */

    /* set up the video capture */
    Capture_Install(CAPTURE_NTSC,CAPTURE_RGB555,CAPTURE_SQCIF);
    Capture_FillBuffs(0x00000000);
    Capture_GetMetrics(&CM);

    /* set up the display */
    Display_SetMode(1024,768,60,DISPLAY_T555,DISPLAY_VIDEO);
    Display_GetMetrics(&DM);
    Display_SetBufferAddresses(0xC0000000,0xC0000000);
    DrawScreen();
    Display_Enable();

```

```
/* try out different capture settings */
while (1) {
    DoCapture(frames,CAPTURE_640x480);
    DoCapture(frames,CAPTURE_512x512);
    DoCapture(frames,CAPTURE_CIF);
    DoCapture(frames,CAPTURE_CIFK);
    DoCapture(frames,CAPTURE_QCIF);
    DoCapture(frames,CAPTURE_SQCIF);
    DoCapture(frames,CAPTURE_QCIF);
    DoCapture(frames,CAPTURE_CIFK);
    DoCapture(frames,CAPTURE_CIF);
    DoCapture(frames,CAPTURE_512x512);
}
}

/*-----*/
void DoCapture(ULONG frames, BYTE scale) {

    int    SemaCt = 0;
    ULONG  CaptureBuff;
    ULONG  DisplayBuff;
    ULONG  dx,dy;
    ULONG  fct;

    /* set the scaling of the capture */
    Capture_SetScaling(scale);
    Capture_FillBufs(0x00000000);
    Capture_GetMetrics(&CM);

    /* initialize the packet transfer table */
    InitPtTable();

    /* draw the display background */
    DrawScreen();

    /* turn on the video */
    Capture_Enable();

    /* video capture-process-display loop */
    for (fct=0; fct<frames; fct++) {

        /* wait for new captured frame */
        TaskWaitSema(CaptureSemaId);

        /* get buffer of most recently captured frame */
        CaptureBuff = Capture_GetBuffer();

        /* tell display driver to toggle buffers next display frame event */
        Display_ToggleBuffers();

        /* wait until display driver has toggled display buffers */
        TaskWaitSema(DisplaySemaId);
    }
}
```

```

    /* get the in-active display buffer */
    DisplayBuff = Display_GetBuffer(DISPLAY_INACTIVE);
    dx = (DM.dx-CM.Rh)/2*DM.bpp/8;
    dy = (DM.dy-CM.Rv)/2*DM.Pitch;

    /* transfer the processed capture buffer to the */
    /* in-active display buffer                      */
    PtTable->SrcStart = CaptureBuff;
    PtTable->DstStart = DisplayBuff+dx+dy;
    PtReqIssue((void *)PtTable,PtSemaId);
    TaskWaitSema(PtSemaId);

    /* unlock the capture buffer obtained from Capture_GetBuffer() */
    Capture_FreeBuffer();
}
Capture_Disable();
}

/*-----*/
void InitPtTable() {

    /* Setup packet transfer table used to transfer the capture buffer to */
    /* the display buffer. The two metrics variables CM and DM must already */
    /* be set before calling this function.                                   */
    PtTable->Next      = (ULONG)PtTable;
    PtTable->Options    = 0x80000000;
    PtTable->SrcStart   = 0x00000000;
    PtTable->DstStart   = 0x00000000;
    PtTable->SrcBCnt    = CM.Rv-1;
    PtTable->SrcACnt    = CM.Rh*CM.Bpp/8;
    PtTable->DstBCnt    = CM.Rv-1;
    PtTable->DstACnt    = CM.Rh*CM.Bpp/8;
    PtTable->SrcCCnt    = 0;
    PtTable->DstCCnt    = 0;
    PtTable->SrcBPitch  = CM.Pitch;
    PtTable->DstBPitch  = DM.Pitch;
    PtTable->SrcCPitch  = 0;
    PtTable->DstCPitch  = 0;
    PtTable->Trans0     = 0;
    PtTable->Trans1     = 0;
    PtTable->Junk1      = 0;
    PtTable->Junk2      = 0;
}

/*-----*/
void DrawScreen() {

    USHORT x,y;
    ULONG  A;
    USHORT color = 0x00007C00;

    Display_FillBufs(0x0007);

```

```

/* draw full frame */
for(x=0; x<DM.dx; x++) {
    Display_SetPixel(x,0,color,DISPLAY_INACTIVE);
    Display_SetPixel(x,DM.dy-1,color,DISPLAY_INACTIVE);
}
for (y=0; y<DM.dy; y++) {
    Display_SetPixel(0,y,color,DISPLAY_INACTIVE);
    Display_SetPixel(DM.dx-1,y,color,DISPLAY_INACTIVE);
}

/* draw scaling boxes */
DrawBox(640,480,color);
DrawBox(512,512,color);
DrawBox(352,288,color);
DrawBox(176,144,color);
DrawBox(128,96,color);
}

/*-----*/
void DrawBox(USHORT dx, USHORT dy, ULONG color) {

    USHORT x,y;

    /* draw box frame */
    for(x=(DM.dx-dx)/2-1; x<(DM.dx-dx)/2+dx; x++) {
        Display_SetPixel(x,(DM.dy-dy)/2-1,color,DISPLAY_INACTIVE);
        Display_SetPixel(x,(DM.dy-dy)/2+dy,color,DISPLAY_INACTIVE);
    }
    for (y=(DM.dy-dy)/2-1; y<(DM.dy-dy)/2+dy; y++) {
        Display_SetPixel((DM.dx-dx)/2-1,y,color,DISPLAY_INACTIVE);
        Display_SetPixel((DM.dx-dx)/2+dx,y,color,DISPLAY_INACTIVE);
    }
}

/*****\
* End of 'captest.c'
*****/

(b) captest.lnk
/*****\
*      Copyright (C) 1995-1996 Texas Instruments Incorporated.
*      All Rights Reserved
*-----
*
*  captest.lnk  -- TMS320C8x MP Linker Command File
*
*****/

-c
-x
-heap 0x00400000

```

```

-stack 0x00010000
-l mp_rts.lib
-l mp_task.lib
-l mp_int.lib
-l mp_ptreq.lib
-l sdbdrvs.lib

MEMORY
{
    PROGMEM : origin=0x80000000 length=0x00800000
}

SECTIONS
{
    .text    :> PROGMEM
    .ptext   :> PROGMEM
    .cinit   :> PROGMEM
    .const   :> PROGMEM
    .switch  :> PROGMEM
    .data    :> PROGMEM
    .bss     :> PROGMEM
    .system  :> PROGMEM
}

/*****\
* End of 'capttest.lnk'
*****/

(c) capttest.bat
@rem # *****\
@rem #      Copyright (C) 1995-1996 Texas Instruments Incorporated.
@rem #                               All Rights Reserved
@rem #-----
@rem #
@rem #  capttest.bat  -- batch file to build project
@rem #
@rem # *****/

@mpc1 -gq capttest.c

@mvplnk -m capttest.map -o capttest.out capttest.obj capttest.lnk

@rem # *****\
@rem # End of 'capttest.bat'
@rem # *****/

```



## A.7 Video Display Test Example

This example tests almost all aspects of the display hardware. You should run this example from a debugger so you can single-step through it.

### Example A-7. *disptest*

#### (a) *disptest.c*

```
/* *****\
 *      Copyright (C) 1995-1996 Texas Instruments Incorporated.
 *      All Rights Reserved
 *-----
 *
 *  disptest.c  -- TMS320C8x MP 'C' Source Code
 *
 *  This is the C file for the disptest module.
 *
 *  !!!NOTE!!!
 *  Some monitors do not support the higher resolutions.  Check your monitor
 *  specifications before attempting to drive it at a high resolution.  Some
 *  monitors do not support resolutions greater than 1024x768.  Remember
 *  also to verify the supported refresh rates.
 * *****/
#include <sdbdrvs.h>

/* *****\
 *      Function Prototypes
 * *****/
void BasicTest();
void ColorTest();
void WindowTest();
void _640x480Test();
void _800x600Test();
void _1024x768Test();
void _1280x1024Test();
void TableTest();
void Fill(ULONG val);
void Delay(ULONG d);

/* *****\
 *      Functions
 * *****/
void main() {

    /* these register settings are mandatory */
    REFCNTL = 0xFFFF0100;
    PTMIN   = 0x00000100;
    PTMAX   = 0x00010000;

    TaskInitTasking();          /* initialize the multitasking executive */
    PtReqInit();                /* needed to allow drivers to use PtReqAlloc() */
}
```

```

Display_Init();          /* called before any other display functions */

if (1) BasicTest();      /* setup simple display mode */
if (1) ColorTest();      /* test all of the color modes */
if (1) _640x480Test();   /* test 640x480 resolution modes */
if (1) _800x600Test();   /* test 800x600 resolution modes */
if (1) _1024x768Test();  /* test 1024x768 resolution modes */

/* !!! only enable the below test when sure monitor can handle it !!! */
if (0) _1280x1024Test(); /* test 1280x1024 resolution modes */

if (1) TableTest();      /* test with custom monitor timing table */
if (1) WindowTest();     /* do some fancy display window programming */

while (1);
}

/*-----*/
void BasicTest() {

    /* Fill up display VRAM with 0x000000FF which is blue in
    /* XRGB format. Set display mode to 640x480 pixels @ 60Hz
    /* refresh rate. True color XRGB 32bpp format. Video output
    /* mode.

    Fill(0x000000FF);
    Display_SetMode(640,480,60,DISPLAY_TXRGB,DISPLAY_VIDEO);
    Display_Enable();
}

/*-----*/
void ColorTest() {

    Display_SetMode(640,480,60, DISPLAY_P8, DISPLAY_VIDEO);
    Display_SetVgaPalette();
    Display_Enable();
    Fill(0x00000000); /* palette offset 0x00 color */
    Fill(0x01010101); /* palette offset 0x01 color */
    Fill(0x02020202); /* palette offset 0x02 color */
    Fill(0x03030303); /* palette offset 0x03 color */
    Fill(0x04040404); /* palette offset 0x04 color */
    Fill(0x05050505); /* palette offset 0x05 color */
    Fill(0x06060606); /* palette offset 0x06 color */
    Fill(0x07070707); /* palette offset 0x07 color */
    Fill(0x08080808); /* palette offset 0x08 color */
    Fill(0x09090909); /* palette offset 0x09 color */
    Fill(0x0A0A0A0A); /* palette offset 0x0A color */
    Fill(0x0B0B0B0B); /* palette offset 0x0B color */
    Fill(0x0C0C0C0C); /* palette offset 0x0C color */
    Fill(0x0D0D0D0D); /* palette offset 0x0D color */
    Fill(0x0E0E0E0E); /* palette offset 0x0E color */
    Fill(0x0F0F0F0F); /* palette offset 0x0F color */
    Display_SetGreyScalePalette();
}

```

```
Display_SetMode(640,480,60, DISPLAY_DXRGB, DISPLAY_VIDEO);
Display_Enable();
Fill(0x00000000); /* BLACK */
Fill(0x000000FF); /* BLUE */
Fill(0x0000FF00); /* GREEN */
Fill(0x0000FFFF); /* CYAN */
Fill(0x00FF0000); /* RED */
Fill(0x00FF00FF); /* MAGENTA */
Fill(0x00FFFF00); /* YELLOW */
Fill(0x00FFFFFF); /* WHITE */

Display_SetMode(640,480,60, DISPLAY_TXRGB, DISPLAY_VIDEO);
Display_Enable();
Fill(0x00000000); /* BLACK */
Fill(0x000000FF); /* BLUE */
Fill(0x0000FF00); /* GREEN */
Fill(0x0000FFFF); /* CYAN */
Fill(0x00FF0000); /* RED */
Fill(0x00FF00FF); /* MAGENTA */
Fill(0x00FFFF00); /* YELLOW */
Fill(0x00FFFFFF); /* WHITE */

Display_SetMode(640,480,60, DISPLAY_DBGRX, DISPLAY_VIDEO);
Display_Enable();
Fill(0x00000000); /* BLACK */
Fill(0xFF000000); /* BLUE */
Fill(0x00FF0000); /* GREEN */
Fill(0xFFFF0000); /* CYAN */
Fill(0x0000FF00); /* RED */
Fill(0xFF00FF00); /* MAGENTA */
Fill(0x00FFFF00); /* YELLOW */
Fill(0xFFFF0000); /* WHITE */

Display_SetMode(640,480,60, DISPLAY_TBGRX, DISPLAY_VIDEO);
Display_Enable();
Fill(0x00000000); /* BLACK */
Fill(0xFF000000); /* BLUE */
Fill(0x00FF0000); /* GREEN */
Fill(0xFFFF0000); /* CYAN */
Fill(0x0000FF00); /* RED */
Fill(0xFF00FF00); /* MAGENTA */
Fill(0x00FFFF00); /* YELLOW */
Fill(0xFFFF0000); /* WHITE */

Display_SetMode(640,480,60, DISPLAY_D565, DISPLAY_VIDEO);
Display_Enable();
Fill(0x00000000); /* BLACK */
Fill(0x001F001F); /* BLUE */
Fill(0x07E007E0); /* GREEN */
Fill(0x07FF07FF); /* CYAN */
Fill(0xF100F100); /* RED */
Fill(0xF11FF11F); /* MAGENTA */
```

```
Fill(0xFFE0FFE0); /* YELLOW */
Fill(0xFFFFFFFF); /* WHITE */

Display_SetMode(640,480,60, DISPLAY_T565, DISPLAY_VIDEO);
Display_Enable();
Fill(0x00000000); /* BLACK */
Fill(0x001F001F); /* BLUE */
Fill(0x07E007E0); /* GREEN */
Fill(0x07FF07FF); /* CYAN */
Fill(0xF100F100); /* RED */
Fill(0xF1FF1F); /* MAGENTA */
Fill(0xFFE0FFE0); /* YELLOW */
Fill(0xFFFFFFFF); /* WHITE */

Display_SetMode(640,480,60, DISPLAY_D555, DISPLAY_VIDEO);
Display_Enable();
Fill(0x00000000); /* BLACK */
Fill(0x001F001F); /* BLUE */
Fill(0x03E003E0); /* GREEN */
Fill(0x03FF03FF); /* CYAN */
Fill(0x7C007C00); /* RED */
Fill(0x7C1F7C1F); /* MAGENTA */
Fill(0x7FE07FE0); /* YELLOW */
Fill(0x7FFF7FFF); /* WHITE */

Display_SetMode(640,480,60, DISPLAY_T555, DISPLAY_VIDEO);
Display_Enable();
Fill(0x00000000); /* BLACK */
Fill(0x001F001F); /* BLUE */
Fill(0x03E003E0); /* GREEN */
Fill(0x03FF03FF); /* CYAN */
Fill(0x7C007C00); /* RED */
Fill(0x7C1F7C1F); /* MAGENTA */
Fill(0x7FE07FE0); /* YELLOW */
Fill(0x7FFF7FFF); /* WHITE */

Display_SetMode(640,480,60, DISPLAY_D664, DISPLAY_VIDEO);
Display_Enable();
Fill(0x00000000); /* BLACK */
Fill(0x000F000F); /* BLUE */
Fill(0x03F003F0); /* GREEN */
Fill(0x03FF03FF); /* CYAN */
Fill(0xFC00FC00); /* RED */
Fill(0xFC0FFC0F); /* MAGENTA */
Fill(0xFFF0FFF0); /* YELLOW */
Fill(0xFFFFFFFF); /* WHITE */

Display_SetMode(640,480,60, DISPLAY_T664, DISPLAY_VIDEO);
Display_Enable();
Fill(0x00000000); /* BLACK */
Fill(0x000F000F); /* BLUE */
Fill(0x03F003F0); /* GREEN */
Fill(0x03FF03FF); /* CYAN */
```

```

Fill(0xFC00FC00); /* RED      */
Fill(0xFC0FFC0F); /* MAGENTA */
Fill(0xFFFF0FFF0); /* YELLOW */
Fill(0xFFFFFFFF); /* WHITE  */

Display_SetMode(640,480,60, DISPLAY_D444, DISPLAY_VIDEO);
Display_Enable();
Fill(0x00000000); /* BLACK   */
Fill(0x00F000F0); /* BLUE    */
Fill(0x0F000F00); /* GREEN   */
Fill(0x0FF00FF0); /* CYAN    */
Fill(0xF000F000); /* RED     */
Fill(0xF0F0F0F0); /* MAGENTA */
Fill(0xFF00FF00); /* YELLOW  */
Fill(0xFFF0FFF0); /* WHITE   */

Display_SetMode(640,480,60, DISPLAY_T444, DISPLAY_VIDEO);
Display_Enable();
Fill(0x00000000); /* BLACK   */
Fill(0x00F000F0); /* BLUE    */
Fill(0x0F000F00); /* GREEN   */
Fill(0x0FF00FF0); /* CYAN    */
Fill(0xF000F000); /* RED     */
Fill(0xF0F0F0F0); /* MAGENTA */
Fill(0xFF00FF00); /* YELLOW  */
Fill(0xFFF0FFF0); /* WHITE   */
}

/*-----*/
void WindowTest() {

    USHORT Rh = 640;
    USHORT Rv = 480;
    USHORT W  = 128;
    float  Xr = Rh/2; /* Rh/2;   */
    float  Yr = 0;    /* Rv/3;   */
    float  Xg = 0;    /* Rh/3;   */
    float  Yg = Rv;    /* 2*Rv/3; */
    float  Xb = Rh;    /* 2*Rh/3; */
    float  Yb = Rv;    /* 2*Rv/3; */
    ULONG  pitch = 4*Rh;
    BYTE   R,G,B;
    float  x,y,Rn,Rr,Rg,Rb;
    ULONG  A,C;

    /* zero out VRAM for visual effect */
    Fill(0x00000000);

    /* set display mode */
    Display_SetMode(Rh,Rv,60, DISPLAY_TXRGB, DISPLAY_VIDEO);
    Display_Enable();

    /* fill display with fancy colors */

```

```

Rn = sqrt(Rh*Rh + Rv*Rv);
for (y=0.0; y<Rv; y+=1.0) {
    for (x=0.0; x<Rh; x+=1.0) {
        Rr = sqrt((x-Xr)*(x-Xr)+(y-Yr)*(y-Yr))/Rn;
        Rg = sqrt((x-Xg)*(x-Xg)+(y-Yg)*(y-Yg))/Rn;
        Rb = sqrt((x-Xb)*(x-Xb)+(y-Yb)*(y-Yb))/Rn;

        R = (BYTE)((1.0-Rr)*255.0);
        G = (BYTE)((1.0-Rg)*255.0);
        B = (BYTE)((1.0-Rb)*255.0);

        C = (R<<16)|(G<<8)|(B);
        A = 0xC0000000+(ULONG)(pitch*y+4*x);

        NOCACHE_ULONG(*(volatile ULONG *)A) = C;
    }
}

x = 0.0;
y = 0.0;

/* wait about a second */
Delay(1000);

/* set display window */
Display_SetWindow(0,0,W,W);

/* move the window around on the screen */
for (y=0; y<Rv-W; y+=8) {
    Display_MoveWindow(0,y);
}

for (x=0; x<Rh-W; x+=8) {
    Display_MoveWindow(x,Rv-W);
}

for (y=Rv-W; y>=0; y-=8) {
    Display_MoveWindow(Rh-W,y);
}

for (x=Rh-W; x>=0; x-=8) {
    Display_MoveWindow(x,0);
}

/* animate window back to full size */
for (x=W; x<=Rh; x+=8)
    Display_SetWindow(0,0,x,W);

for (y=W; y<=Rv; y+=8)
    Display_SetWindow(0,0,Rh,y);

/* wait about a second */

```

```
    Delay(1000);
}

/*-----*/
void _640x480Test() {

    Fill(0xFFFFFFFF);

    Display_SetMode(640,480,60, DISPLAY_P8,    DISPLAY_VIDEO); Display_Enable();
    Display_SetMode(640,480,60, DISPLAY_DXRGB,DISPLAY_VIDEO); Display_Enable();
    Display_SetMode(640,480,60, DISPLAY_DBGRX,DISPLAY_VIDEO); Display_Enable();
    Display_SetMode(640,480,60, DISPLAY_D565,  DISPLAY_VIDEO); Display_Enable();
    Display_SetMode(640,480,60, DISPLAY_D555,  DISPLAY_VIDEO); Display_Enable();
    Display_SetMode(640,480,60, DISPLAY_D664,  DISPLAY_VIDEO); Display_Enable();
    Display_SetMode(640,480,60, DISPLAY_D444,  DISPLAY_VIDEO); Display_Enable();
    Display_SetMode(640,480,60, DISPLAY_TXRGB, DISPLAY_VIDEO); Display_Enable();
    Display_SetMode(640,480,60, DISPLAY_TBGRX, DISPLAY_VIDEO); Display_Enable();
    Display_SetMode(640,480,60, DISPLAY_T565,  DISPLAY_VIDEO); Display_Enable();
    Display_SetMode(640,480,60, DISPLAY_T555,  DISPLAY_VIDEO); Display_Enable();
    Display_SetMode(640,480,60, DISPLAY_T664,  DISPLAY_VIDEO); Display_Enable();
    Display_SetMode(640,480,60, DISPLAY_T444,  DISPLAY_VIDEO); Display_Enable();

    Display_SetMode(640,480,72, DISPLAY_P8,    DISPLAY_VIDEO); Display_Enable();
    Display_SetMode(640,480,72, DISPLAY_DXRGB,DISPLAY_VIDEO); Display_Enable();
    Display_SetMode(640,480,72, DISPLAY_DBGRX,DISPLAY_VIDEO); Display_Enable();
    Display_SetMode(640,480,72, DISPLAY_D565,  DISPLAY_VIDEO); Display_Enable();
    Display_SetMode(640,480,72, DISPLAY_D555,  DISPLAY_VIDEO); Display_Enable();
    Display_SetMode(640,480,72, DISPLAY_D664,  DISPLAY_VIDEO); Display_Enable();
    Display_SetMode(640,480,72, DISPLAY_D444,  DISPLAY_VIDEO); Display_Enable();
    Display_SetMode(640,480,72, DISPLAY_TXRGB, DISPLAY_VIDEO); Display_Enable();
    Display_SetMode(640,480,72, DISPLAY_TBGRX, DISPLAY_VIDEO); Display_Enable();
    Display_SetMode(640,480,72, DISPLAY_T565,  DISPLAY_VIDEO); Display_Enable();
    Display_SetMode(640,480,72, DISPLAY_T555,  DISPLAY_VIDEO); Display_Enable();
    Display_SetMode(640,480,72, DISPLAY_T664,  DISPLAY_VIDEO); Display_Enable();
    Display_SetMode(640,480,72, DISPLAY_T444,  DISPLAY_VIDEO); Display_Enable();
}

/*-----*/
void _800x600Test() {

    Fill(0xFFFFFFFF);

    Display_SetMode(800,600,60, DISPLAY_P8,    DISPLAY_VIDEO); Display_Enable();
    Display_SetMode(800,600,60, DISPLAY_DXRGB,DISPLAY_VIDEO); Display_Enable();
    Display_SetMode(800,600,60, DISPLAY_DBGRX,DISPLAY_VIDEO); Display_Enable();
    Display_SetMode(800,600,60, DISPLAY_D565,  DISPLAY_VIDEO); Display_Enable();
    Display_SetMode(800,600,60, DISPLAY_D555,  DISPLAY_VIDEO); Display_Enable();
    Display_SetMode(800,600,60, DISPLAY_D664,  DISPLAY_VIDEO); Display_Enable();
    Display_SetMode(800,600,60, DISPLAY_D444,  DISPLAY_VIDEO); Display_Enable();
    Display_SetMode(800,600,60, DISPLAY_TXRGB,DISPLAY_VIDEO); Display_Enable();
    Display_SetMode(800,600,60, DISPLAY_TBGRX,DISPLAY_VIDEO); Display_Enable();
    Display_SetMode(800,600,60, DISPLAY_T565,  DISPLAY_VIDEO); Display_Enable();
    Display_SetMode(800,600,60, DISPLAY_T555,  DISPLAY_VIDEO); Display_Enable();
    Display_SetMode(800,600,60, DISPLAY_T664,  DISPLAY_VIDEO); Display_Enable();
    Display_SetMode(800,600,60, DISPLAY_T555,  DISPLAY_VIDEO); Display_Enable();
}
```

```

    Display_SetMode(800,600,60, DISPLAY_T664, DISPLAY_VIDEO); Display_Enable();
    Display_SetMode(800,600,60, DISPLAY_T444, DISPLAY_VIDEO); Display_Enable();
}

/*-----*/
void _1024x768Test() {

    Fill(0xFFFFFFFF);

    Display_SetMode(1024,768,60, DISPLAY_P8, DISPLAY_VIDEO); Display_Enable();
    Display_SetMode(1024,768,60, DISPLAY_D565,DISPLAY_VIDEO); Display_Enable();
    Display_SetMode(1024,768,60, DISPLAY_D555,DISPLAY_VIDEO); Display_Enable();
    Display_SetMode(1024,768,60, DISPLAY_D664,DISPLAY_VIDEO); Display_Enable();
    Display_SetMode(1024,768,60, DISPLAY_D444,DISPLAY_VIDEO); Display_Enable();
    Display_SetMode(1024,768,60, DISPLAY_T565,DISPLAY_VIDEO); Display_Enable();
    Display_SetMode(1024,768,60, DISPLAY_T555,DISPLAY_VIDEO); Display_Enable();
    Display_SetMode(1024,768,60, DISPLAY_T664,DISPLAY_VIDEO); Display_Enable();
    Display_SetMode(1024,768,60, DISPLAY_T444,DISPLAY_VIDEO); Display_Enable();

    Display_SetMode(1024,768,70, DISPLAY_P8, DISPLAY_VIDEO); Display_Enable();
    Display_SetMode(1024,768,70, DISPLAY_D565,DISPLAY_VIDEO); Display_Enable();
    Display_SetMode(1024,768,70, DISPLAY_D555,DISPLAY_VIDEO); Display_Enable();
    Display_SetMode(1024,768,70, DISPLAY_D664,DISPLAY_VIDEO); Display_Enable();
    Display_SetMode(1024,768,70, DISPLAY_D444,DISPLAY_VIDEO); Display_Enable();
    Display_SetMode(1024,768,70, DISPLAY_T565,DISPLAY_VIDEO); Display_Enable();
    Display_SetMode(1024,768,70, DISPLAY_T555,DISPLAY_VIDEO); Display_Enable();
    Display_SetMode(1024,768,70, DISPLAY_T664,DISPLAY_VIDEO); Display_Enable();
    Display_SetMode(1024,768,70, DISPLAY_T444,DISPLAY_VIDEO); Display_Enable();
}

/*-----*/
void _1280x1024Test() {

    Fill(0xFFFFFFFF);

    Display_SetMode(1280,1024,60, DISPLAY_P8, DISPLAY_VIDEO); Display_Enable();
}

/*-----*/

/* custom monitor timing table */
DISPLAY_MT CustomTable[] = {
/* A Rh Rv Fv Fh Fd Ths Thbp Tvs Tvbp Sh Sv */
/* pels pels Hz kHz MHz usec usec usec usec */
{ 1, 640, 480, 60.0, 31.4, 25.2, 1.00, 2.00, 100.0, 600.0, 0, 0},
{ 1, 800, 600, 60.0, 37.8, 40.0, 1.00, 2.00, 100.0, 400.0, 0, 0},
{ 1, 1024, 768, 60.0, 48.3, 65.0, 1.00, 2.75, 100.0, 350.0, 0, 0},
{ 1, 1152, 864, 60.0, 54.9, 82.0, 1.00, 2.30, 100.0, 500.0, 0, 0},
{-1, 0000, 0000, 00.0, 00.0, 000.0, 0.00, 0.00, 000.0, 000.0, 0, 0}
};

/* 1152x864 is a custom display mode */

```



```

void TableTest() {
    Fill(0x001F001F); /* blue for T555 */

    /* install the new timing table */
    Display_InstallTimingTable(CustomTable);

    /* Display_SetMode() will use the new timing table for now on */
    Display_SetMode(1152,864,60, DISPLAY_T555, DISPLAY_VIDEO);
    Display_Enable();

    /* wait about 5 seconds */
    Delay(5000);
}

/*-----*/
void Fill(ULONG val) {
    ULONG A;

    /* Fills all of VRAM with val */
    for (A=0xC0000000; A<0xC0200000; A+=4)
        NOCACHE_ULONG(*(ULONG *)A) = val;
}

/*-----*/
void Delay(ULONG d) {
    ULONG i;

    /* about 1ms per d */
    for (i=d*4500; i>0; i--);
}

/*****\
* End of 'disptest.c'
*****/

(b) disptest.lnk
/*****\
*      Copyright (C) 1995-1996 Texas Instruments Incorporated.
*                  All Rights Reserved
*-----
*
* disptest.lnk  -- TMS320C8x MP Linker Command File
*
*****/

-c
-x
-heap 0x00100000
-stack 0x00010000

```

```

-l mp_rts.lib
-l mp_task.lib
-l mp_int.lib
-l mp_ptreq.lib
-l sdbdrvs.lib

MEMORY
{
    PROGMEM : origin=0x80000000 length=0x00800000
}
SECTIONS
{
    .text    :> PROGMEM
    .ptext   :> PROGMEM
    .cinit   :> PROGMEM
    .const   :> PROGMEM
    .switch  :> PROGMEM
    .data    :> PROGMEM
    .bss     :> PROGMEM
    .system  :> PROGMEM
}

/*****\
* End of 'disptest.lnk'
*****/

(c) disptest.bat
@rem # *****\
@rem #      Copyright (C) 1995-1996 Texas Instruments Incorporated.
@rem #      All Rights Reserved
@rem #-----
@rem #
@rem # disptest.bat -- batch file to build project
@rem #
@rem # *****/

@mpc1 -gq disptest.c

@mvplnk -m disptest.map -o disptest.out disptest.obj disptest.lnk

@rem # *****\
@rem # End of 'disptest.bat'
@rem # *****/

```

# Shared Data Types and Macros

---

---

---

This chapter contains two header files, <sdbdrvs.h> and <hsdbdrvs.h>, which define shared data types and macros used among the driver modules. The 'C80 driver modules all share the data types and macros defined in <sdbdrvs.h>; the host communications driver modules share the data types and macros defined in <hsdbdrvs.h>.

| Topic   | Page |
|---|------|
| B.1 TMS320C80 API Library Header File <sdbdrvs.h> ..... | B-2  |
| B.2 Host API Library Header File <hsdbdrvs.h> .....     | B-4  |

## B.1 TMS320C80 API Library Header File <sdbdrvs.h>

The data types and macros shared among the 'C80 driver modules, audio, display, capture, and server, are defined in sdbdrvs.h. Following is the contents of this header file. A 'C80 application only needs to include sdbdrvs.h to use any of the driver modules.

```
/* *****\
*          Copyright (C) 1996 Texas Instruments Incorporated.
*          All Rights Reserved
*-----
*
* sdbdrvs.h  -- TMS320C8x MP 'C' Header File
*
* This is the header file for the entire sdb C80 driver library.
*
\*****/
#ifndef _SDBDRVS_H
#define _SDBDRVS_H

/* include commonly used header files */
#include <mvp.h>
#include <mvp_hw.h>
#include <task.h>
#include <mp_ptreq.h>
#include <stdlib.h>
#include <math.h>

/* makes for shorter and neater code */
typedef unsigned char  BYTE;
typedef unsigned short USHORT;
typedef unsigned long  ULONG;
typedef unsigned char  BOOL;

/* simple DIM to DIM packet transfer table structure */
typedef struct {
    ULONG  Next;
    ULONG  Options;
    ULONG  SrcStart;
    ULONG  DstStart;
    short  SrcBCnt;
    short  SrcACnt;
    short  DstBCnt;
    short  DstACnt;
    long   SrcCCnt;
    long   DstCCnt;
    long   SrcBPitch;
    long   DstBPitch;
    long   SrcCPitch;
    long   DstCPitch;
    ULONG  Trans0;
    ULONG  Trans1;
```

```
    ULONG   Junk1;
    ULONG   Junk2;
} PT;

#ifndef TRUE
#define TRUE 1
#endif

#ifndef FALSE
#define FALSE 0
#endif

#ifndef NULL
#define NULL 0
#endif

/* include the driver header files */
/* note that _SDBAPI_ is only defined from within a driver module */
#if !defined _SDBAPI_
#include <audio.h>
#include <display.h>
#include <capture.h>
#include <sserver.h>
#endif

#endif /* _SDBDRVS_H */
/*****\
* End of 'sdbdrvs.h'
\*****/
```

## B.2 Host API Library Header File <hsdbdrv.h>

The data types and macros shared among the host client module and its sub-modules are defined in `hsdbdrv.h`. Following is the contents of this header file. A host application only needs to include `hsdbdrv.h` to use the driver module.

```
/*
 * Copyright (C) 1996 Texas Instruments Incorporated.
 * All Rights Reserved
 *-----
 *
 * hsdbdrv.h -- PC 'C' Header File
 *
 * This is the header file for the entire host library.
 */
#define _HSDBDRVS_H

/* include some commonly used header files */
#include <windows.h>
#include <stdio.h>

/* makes for shorter and neater code */
typedef unsigned char BYTE;
typedef unsigned short USHORT;
typedef unsigned long ULONG;
typedef int BOOL; /* BOOL is an 'int' in "windef.h" */

#ifndef TRUE
#define TRUE 1
#endif

#ifndef FALSE
#define FALSE 0
#endif

#ifndef NULL
#define NULL 0
#endif

/* include the driver header file */
#if !defined _HOSTAPI_
#include <hclient.h>
#endif

#endif /* _HSDBDRVS_H */
/* End of 'hsdbdrv.h' */
```

# API Functions With Arguments and Return Types

---

---

---

This appendix lists all the API functions in alphabetical order by function name, including their arguments and return types. Use this list as a function protocol list and as a reminder of argument order.

| Function   | Page |
|--|------|
| void <b>Audio_CaptureToMemory</b> (AUDIO_PTR *P, ULONG NumBufs); .....   | 3-10 |
| BYTE <b>Audio_CodecStat</b> (); .....  | 3-12 |
| void <b>Audio_Disable</b> (); .....  | 3-13 |
| void <b>Audio_Enable</b> (); .....   | 3-14 |
| USHORT <b>Audio_FifoStat</b> (); .....   | 3-15 |
| void <b>Audio_FillBufs</b> (USHORT val); .....   | 3-16 |
| BOOL <b>Audio_GetCaptureBufs</b> (AUDIO_PTR *P); .....   | 3-17 |
| void <b>Audio_GetCodecRegs</b> (BYTE *CodecRegs); .....  | 3-18 |
| void <b>Audio_GetFifoRegs</b> (USHORT *FifoRegs); .....  | 3-19 |
| void <b>Audio_GetMetrics</b> (AUDIO_MET *M); .....   | 3-20 |
| BOOL <b>Audio_GetPlaybackBufs</b> (AUDIO_PTR *P); .....  | 3-21 |
| BOOL <b>Audio_Init</b> (); .....   | 3-22 |
| BOOL <b>Audio_Install</b> (BYTE Mode, BYTE Format, BOOL Stereo, float Fs,<br>BYTE BlockCt, BYTE BlockSz, BYTE BuffCt); ..... | 3-23 |
| long <b>Audio_InstallSema</b> (long SemaId); .....   | 3-25 |
| void <b>Audio_PioIn</b> (void *left, void *right); .....   | 3-26 |
| void <b>Audio_PioOut</b> (void *left, void *right); .....  | 3-27 |
| void <b>Audio_PioTest</b> (BYTE Format, BOOL Stereo, float Fs, ULONG Ct); .....  | 3-28 |
| void <b>Audio_PlaybackFromMemory</b> (AUDIO_PTR *P, ULONG NumBufs); .....  | 3-29 |
| void <b>Audio_ProgramAux1</b> (BOOL lmute, BOOL rmute, float lg, float rg); .....  | 3-31 |
| void <b>Audio_ProgramDacs</b> (BOOL lmute, BOOL rmute, float lg, float rg); .....  | 3-32 |
| void <b>Audio_ProgramDigitalMix</b> (BOOL enable, float gain); .....   | 3-33 |
| void <b>Audio_ProgramInputs</b> (BYTE lsrc, BYTE rsrc, float lg, float rg); .....  | 3-34 |
| BOOL <b>Audio_SetBufferIndexes</b> (BYTE Capp, BYTE Cisir, BYTE Papp, BYPE Pisir); .....                                     | 3-35 |
| void <b>Audio_SetSampleRate</b> (float Fs); .....  | 3-36 |
| void <b>Audio_UnInstall</b> (); .....  | 3-37 |
| BOOL <b>Capture_CardPresent</b> (); .....  | 5-9  |
| void <b>Capture_Disable</b> (); .....  | 5-10 |
| void <b>Capture_Enable</b> (); .....   | 5-11 |
| void <b>Capture_FillBufs</b> (ULONG val); .....  | 5-12 |
| void <b>Capture_FreeBuffer</b> (); .....   | 5-13 |
| ULONG <b>Capture_GetBuffer</b> (); .....   | 5-14 |
| void <b>Capture_GetDecoderRegs</b> (BYTE *DR); .....   | 5-15 |
| void <b>Capture_GetMetrics</b> (CAPTURE_MET *M); .....   | 5-16 |
| void <b>Capture_GetScalerRegs</b> (BYTE *SR); .....  | 5-17 |



| Function  | Page |
|---|------|
| BOOL <b>Capture_Init</b> (); .....  | 5-18 |
| BOOL <b>Capture_Install</b> (BYTE InFormat, BYTE OutFormat, BYTE Scaling); .....                        | 5-19 |
| long <b>Capture_InstallSema</b> (long SemaId); .....  | 5-21 |
| void <b>Capture_SetInputSource</b> (BYTE InSrc); .....  | 5-22 |
| BOOL <b>Capture_SetScaling</b> (BYTE Scaling); .....  | 5-23 |
| void <b>Capture_UnInstall</b> (); .....   | 5-24 |
| CLIENT_STAT <b>Client_Boot</b> (char *coffname, ULONG EntryPoint, USHORT Flags,<br>FILE *Output); ..... | 6-12 |
| CLIENT_STAT <b>Client_ClearConfigBit</b> (ULONG bit); .....   | 6-13 |
| CLIENT_STAT <b>Client_Close</b> (); .....   | 6-14 |
| BOOL <b>Client_Init</b> (); .....   | 6-15 |
| CLIENT_STAT <b>Client_IssueCmnd</b> (ULONG *Cmnd, long TimeOut); .....                                  | 6-16 |
| CLIENT_STAT <b>Client_Open</b> (); .....  | 6-17 |
| CLIENT_STAT <b>Client_ReadConfig</b> (ULONG *Val); .....  | 6-18 |
| CLIENT_STAT <b>Client_ReadConfigBit</b> (ULONG bit); .....  | 6-19 |
| CLIENT_STAT <b>Client_ReadDataFifo</b> (ULONG *Block, ULONG ct, long timeout); .....                    | 6-20 |
| CLIENT_STAT <b>Client_ReadIo</b> (USHORT Address, ULONG *Val); .....                                    | 6-21 |
| CLIENT_STAT <b>Client_ReadMailbox</b> (ULONG *Val, long timeout); .....                                 | 6-22 |
| CLIENT_STAT <b>Client_Reset</b> (USHORT Flags, FILE *Output); .....                                     | 6-23 |
| CLIENT_STAT <b>Client_SetConfigBit</b> (ULONG bit); .....   | 6-24 |
| CLIENT_STAT <b>Client_SetSwapping</b> (USHORT sw); .....  | 6-25 |
| CLIENT_STAT <b>Client_Sync</b> (long timeout); .....  | 6-26 |
| CLIENT_STAT <b>Client_WriteConfig</b> (ULONG Val); .....  | 6-27 |
| CLIENT_STAT <b>Client_WriteDataFifo</b> (ULONG *Block, ULONG ct, long timeout); .....                   | 6-28 |
| CLIENT_STAT <b>Client_WriteIo</b> (USHORT Address, ULONG Val); .....                                    | 6-29 |
| CLIENT_STAT <b>Client_WriteMailbox</b> (ULONG Val, long timeout); .....                                 | 6-30 |
| void <b>Display_Disable</b> (); .....   | 4-11 |
| void <b>Display_Enable</b> (); .....  | 4-12 |
| void <b>Display_FillBufs</b> (ULONG val); .....   | 4-13 |
| ULONG <b>Display_GetBuffer</b> (BYTE buffid); .....   | 4-14 |
| void <b>Display_GetMetrics</b> (DISPLAY_MET *M); .....  | 4-15 |
| void <b>Display_GetTvpRegs</b> (BYTE *R); .....   | 4-16 |
| BOOL <b>Display_Init</b> (); .....  | 4-17 |
| long <b>Display_InstallSema</b> (long SemaId); .....  | 4-18 |
| void <b>Display_InstallTimingTable</b> (DISPLAY_MT *Table); .....                                       | 4-19 |

| Function   | Page |
|--|------|
| void <b>Display_MoveWindow</b> (USHORT x, USHORT y); .....   | 4-20 |
| void <b>Display_ReadPalette</b> (BYTE *R, BYTE *G, BYTE *B); .....   | 4-21 |
| void <b>Display_SetBufferAddresses</b> (ULONG Buff1, ULONG Buff2); .....   | 4-22 |
| float <b>Display_SetDotClock</b> (float Fd); .....   | 4-23 |
| void <b>Display_SetGreyScalePalette</b> (); .....  | 4-24 |
| BOOL <b>Display_SetMode</b> (USHORT Rh, USHORT Rv, float Fv, char *PixFmt,<br>BYTE Output); .....  | 4-25 |
| void <b>Display_SetOverlayParams</b> (short adx, short ady, short bdx, short bdy,<br>float dThbp, float dTvbp, float Fd, float dFd); ..... | 4-27 |
| void <b>Display_SetPaletteAddress</b> (BYTE ad); .....   | 4-29 |
| void <b>Display_SetPitch</b> (ULONG pitch); .....  | 4-30 |
| void <b>Display_SetPixel</b> (USHORT x, USHORT y, ULONG val, BYTE buffid); .....   | 4-31 |
| void <b>Display_SetSyncPolarities</b> (BYTE Sh, BYTE Sv, BYTE Pen); .....  | 4-32 |
| void <b>Display_SetVgaPalette</b> (); .....  | 4-33 |
| void <b>Display_SetWindow</b> (USHORT x, USHORT y, USHORT dx, USHORT dy); .....  | 4-34 |
| void <b>Display_ToggleBuffers</b> (); .....  | 4-35 |
| BYTE <b>Display_TvpRegIn</b> (BYTE reg); .....   | 4-36 |
| void <b>Display_TvpRegOut</b> (BYTE reg, BYTE val); .....  | 4-37 |
| void <b>Display_WaitEndOfFrame</b> (); .....   | 4-38 |
| void <b>Display_WritePalette</b> (BYTE R, BYTE G, BYTE B); .....   | 4-39 |
| void <b>Server_Done</b> (); .....  | 6-31 |
| void <b>Server_GetCmnd</b> (ULONG *Cmnd); .....  | 6-32 |
| BOOL <b>Server_Init</b> (); .....  | 6-33 |
| void <b>Server_Install</b> (); .....   | 6-34 |
| long <b>Server_InstallSema</b> (long SemaId); .....  | 6-35 |
| long <b>Server_ReadDataFifo</b> (ULONG *Dst, USHORT Length); .....   | 6-36 |
| void <b>Server_Sync</b> (); .....  | 6-37 |
| void <b>Server_UnInstall</b> (); .....   | 6-38 |
| void <b>Server_WriteDataFifo</b> (ULONG *Src, USHORT Length); .....  | 6-39 |

# Glossary

---

---

---

## A

**A-Law companding:** *See companded.*

**active time:** (vs. blanking) The time intervals of a display frame that are not in blanking. The time intervals in which pixels are displayed.

**ADC:** *Analog-to-digital converter.* A device that converts a continuously varying signal (analog) to a signal represented by a series of numbers (digital).

**analog mixing:** The mixing together of two analog signals. The multiplexing of two analog signals into one.

**API:** *Application programming interface.* Used for proprietary application programs to interact with communications software or to conform to protocols from another vendor's product.

**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.

**autocalibration:** Automatic adjustment of a device so that the output is within a specific range for particular values of the input.

## B

**back porch:** The interval of the video waveform between the end of synchronization and the corresponding blanking pulse. The horizontal back porch is specified as an integral number of FCLK periods; the vertical back porch is specified as an integral number of lines (halflines for interlaced mode). *See also front porch.*

**bit plane:** A bit storage array (plane) used to store a particular bit of each pixel of an image. The 0th bit of each pixel is stored in bit plane 0, the first bit of each pixel is stored in bit plane 1, and so on.

**blanking:** Extinguishing the scanning beam during horizontal and vertical retrace periods. See also *horizontal blanking*, *vertical blanking*.

**blanking area:** The area of a display that is not active but rather blanked. No pixels are displayed in the blanking area. Vertical and horizontal retrace occur during blanking.

**BPP:** *Bits per pixel*. The number of bits used to represent the color value of each pixel in a digitized image.

## C

**capture mode:** A mode of the audio subsystem in which DMA transfers read audio data that has been captured by the audio codec.

**chrominance:** The NTSC or PAL video signal contains two pieces that make up what you see on the screen: the black and white part (luminance) and the color part. Chrominance is the color part. See also *luminance*.

**codec:** *Coder-decoder*, or *compression/decompression*, typically of video or audio data.

**COFF:** *Common object file format*. A file format used by the 'C8x for compiler and linker output files. A COFF file is organized into sections by the compiler.

**companded:** *Compressed and expanded*. 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 used in Europe (A-Law) and the United States ( $\mu$ -Law).

**composite area:** The signal generated by the frame timers that can be used to define a special area, such as an overscan boundary. This signal acts identically in both interlaced and noninterlaced modes, defining a purely rectangular region.

**composite video:** See *CVBS*.

**CVBS:** *Composite video*. Signal that carries video picture information for color, brightness, and synchronization signals for both horizontal and vertical scans.

**D**

**DAC:** *Digital-to-analog converter.* A device that converts a signal represented by a series of numbers (digital) to a continuously varying signal (analog).

**deassert:** To make a digital logic device pin inactive. If the pin is active low, then a high voltage on the pin deasserts it. If the pin is active high, then a low voltage deasserts it.

**debugger:** A window-oriented software interface that helps you to detect and fix errors in programs running on a 'C8x.

**digital mixing:** The mixing together of two digital signals into one. The algebraic sum of two digital signals.

**digital signal processor:** See *DSP*.

**digitizer:** The part of the video capture front end that converts the analog video signal into a digital signal to be decoded.

**display mode:** The mode in which pixels are shown on a display device and also the resolution of a display.

**display window:** Reducing the active area of a display creates a display window.

**DMA mode:** *Direct memory access mode.* A mode of the audio subsystem in which DMA transfers read audio data that has been captured by the audio codec or in which DMA transfers supply audio data for playback. DMA capture and playback are not possible simultaneously.

**dot clock:** The clock that cycles the rate at which video data is output to a display monitor.

**DRAM:** *Dynamic random-access memory.* Memory typically used for external memory. A special memory circuit that is dynamic in nature; it requires each bit of information to be refreshed, or restored to its programmed state, on a periodic basis to maintain valid data.

**DSP:** *Digital signal processor.* A processor used for high speed data manipulations of audio, video, graphical, or image information.

**E**

**event pin:** A pin on the SDB's interrupt controller that triggers an event when asserted.

**externally initiated packet transfer:** See *XPT*.

## F

**FCLK:** *Frame clock.* The clock that controls the internal video logic of the video controller's frame timers.

**FIFO:** *First in, first out.* 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 which are not synchronized; that is, 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 will become either full (blocking the sender) or empty (blocking the receiver).

**FIFO flag:** Indicator that gets set or cleared depending on the state of the FIFO.

**FIFO status register:** A register located within a FIFO device used to store status information regarding the device.

**flag:** A variable or quantity that can take on one of two values. A bit, particularly one that is used to indicate one of two outcomes or is used to control which of two things is to be done.

**flag offsets:** The offset into a FIFO memory device that determines when FIFO flags get set or cleared.

**frame:** The screen image output during a single vertical sweep.

**frame timer:** In the video controller (VC), a timer that provides video timing control and indicates to the serial register transfer (SRT) controller when an SRT is necessary.

**front porch:** The interval of a video waveform between the start of blanking and the corresponding sync pulse. The horizontal front porch is specified as an integral number of FCLK periods; the vertical front porch is specified as an integral number of lines (half-lines for interlaced mode). See also *back porch*.

**G**

**gain stage:** That portion of a circuit which imposes a gain onto a signal. Also that portion of an algorithm that imposes a gain onto a digital signal.

**grayscale:** Or *greyscale*. A range of accurately known shades of gray printed out for use in calibrating those shades on a display or printer. In graphics, composed of discrete shades of gray. For displays, a color format in which each pixel is determined by an 8-bit value. This value maps to RGB space with the red, green, and blue components all taking on the 8-bit value. The result is pixels which can range from black, to gray, to pure white.

**H**

**horizontal blanking:** A bidirectional timing signal that enables or disables pixel capture and display. Horizontal blanking occurs once per line and its pulse width is defined as an integral number of FCLK periods. See also *blanking*.

**horizontal sync:** The portion of the composite video signal that tells the receiver where to place the image in the left-to-right dimension. The horizontal sync pulse tells the receiving system where the beginning of the new scan line is. See also *vertical sync*.

**I**

**IEEE standard 1149.1-1990:** See *JTAG*.

**IEEE-754 floating point unit:** The floating point math unit contained in the core of the TMS320C8x's master RISC processor.

**interlaced mode:** A video mode in which each frame consists of two vertical fields. One field displays odd horizontal lines, and the other field displays even horizontal lines. In effect, the number of transmitted pictures is doubled, thus reducing flicker.

**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:** See *ISR*.

**ISR:** *Interrupt service routine*. A module of code that is executed in response to a hardware or software interrupt.

## J

**JPEG standard:** *Joint Photographic Experts Group standard.* A standard used for compressed still-picture data.

**JTAG:** *Joint Test Action Group.* 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

**kHz:** *Kilohertz.*

## L

**line dropping:** The process of eliminating lines of video from an image to downscale the image vertically.

**look-up table:** A table, used during scan conversions of a digital image, that converts color-map addresses into the actual color values displayed.

**luminance:** The NTSC or PAL video signal contains two pieces that make up what you see on the screen: the black and white part and the color part (chrominance). The black and white part is called the luminance. See also *chrominance*.

## M

**μ-Law companding:** *See companded.*

**master processor:** *See MP.*

**member:** An element or variable of a structure, union, or enumeration.

**merge mode:** A serial register transfer (SRT) mode for the video controller (VC) during which an image is capture and stored in memory. Memory locations not corresponding to the captured image are preserved. See also *capture mode*, *display mode*.

**metric parameters:** A set of parameters that define state dimensions and attributes for the audio subsystem, display subsystem, or video capture subsystem.



**MHz:** *Megahertz.*

**monitor timing (parameters):** Parameters that the display API uses to determine what signal rates are needed to drive a monitor.

**mono mode:** A mode of the audio codec in which only one channel of audio exists.

**MP:** *Master processor.* A general-purpose RISC processor that coordinates the activity of the other processors on the 'C8x. The 'C8x includes an IEEE1-754 floating-point hardware unit.

**MPEG standard:** *Moving Picture Experts Group standard.* A proposed standard for compressed video data.

**multiplexing:** A process of transmitting more than one set of signals at a time over a single wire or communications link.

**multisync monitor:** A monitor that adjusts itself to the horizontal and vertical synchronization rate of the video signal. A multisync monitor can be used with a variety of video adapters.

**mutual exclusion:** A collection of techniques for sharing resources so that different uses do not conflict and cause unwanted interactions. One of the most commonly used techniques for mutual exclusion is the semaphore. See also *semaphore*.

## N

**noninterlaced graphics mode:** A mode for the video controller (VC) in which each frame consists of a single vertical field. A method of scanning out a video display where all of the lines in the frame are scanned out sequentially, one right after the other. Also called *progressive scan*.

**NTSC:** *National Television Standards Committee.* A color television broadcast standard wherein the image consists of a format that has 525 scan lines; a field frequency of 60 Hz; a broadcast bandwidth of 4 MHz; a line frequency of 15.75 kHz; a frame frequency of one-thirtieth of a second; and a color subcarrier frequency of 3.58 MHz. See also *PAL*.

## O

**overlay mode:** Mixed video mode. The input from the VGA pass-through cable is mixed with the RAMDAC output to form video overlaid onto VGA.

**P**

**PAL:** *Phase alternation line.* A European deviation of the standard U.S. television NTSC signal; the format is 625 lines and a 50-Hz frequency. See also *NTSC*.

**parallel processor:** See *PP*.

**PCA:** *Printed-circuit assembly.* A printed-circuit board on which separately manufactured component parts have been installed in an electrical circuit that performs a defined function.

**PCI:** *Peripheral component interconnect.* High-speed local bus that supports data-transfer speeds of up to 132M bytes per second at 33 MHz.

**PCM:** *Pulse code modulation.*

**PIO mode:** *Programmed input/output mode.* A mode of the audio codec in which DMA is not use; rather, samples are directly read from or written to the PIO port.

**pitch:** The number of bytes between the start of one line to the start of the next line in a frame of video.

**pixel:** One picture element (pel).

**pixel dropping:** The process of removing pixels from a line of video to down-scale that line.

**playback mode:** A mode of the audio subsystem in which DMA transfers supply audio data for playback.

**porch:** The portion of a video display signal that corresponds to the blanking interval on either side of a horizontal or vertical sync pulse. The terms front porch and back porch refer to the blanking intervals that precede and follow, respectively, the sync pulse. See also *back porch*, *front porch*.

**PP:** *Parallel processor.* The 'C8x's advanced digital signal processor that is used for video compression/demcompression (P×64 or MPEG), still-image compression/decompression (JPEG), 2-D and 3-D graphic functions such as line draw, trapezoid fill, antialiasing, and a variety of high-speed integer operations on image data. A 'C8x single-chip multiprocessor device may contain from one to eight PPs, depending on the device version.

**progressive scan:** See *noninterlaced graphics mode*.

**Q**

**quantization error:** The error resulting from converting an analog signal into a digital signal due to the fact that a digital signal can only have discrete values whereas an analog signal may take on any value within dynamic range of the signal.

**quantum levels:** When a signal can only take on certain discrete values, these values are referred to as quantum levels.

**R**

**RAMDAC:** *Random-access memory digital-to-analog converter.* Used to convert digital RGB (red-green-blue) information to analog signals that drive a display.

**raster:** The series of scan lines that comprise a television picture or a computer's display. A raster line is the same as a scan line, which is an individual sweep across the face of the display by the electron beam that makes the picture.

**refresh rate:** The speed with which a video source redisplay the screen.

**RISC:** *Reduced instruction set computer.* 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.

**S**

**sample rate:** The rate at which the audio codec samples audio data. Usually specified in hertz (samples per second).

**SDB:** *TMS320C8x software development board.*

**semaphore:** A classic method for restricting access to shared resources (for example, storage) in a multiprocessing environment. A semaphore is a protected variable (or abstract data type) that can be accessed only by certain operations for testing and incrementing the value of the variable.

**semiomnipresent pixel:** A pixel that appears to be at two locations on the screen at once in a video display.

**serial register transfer:** See *SRT controller*.

**skew:** Time differences in multiple clock signals based on physical distances between the origin of the signals and their destinations. Switching delays caused by gates in the logic.

**S-VHS:** *Super VHS (vertical helical scan)*. Similar to the VHS video recording standard, except that the chrominance and luminance data are treated as components that provide higher quality video.

**SRT controller:** *Serial register transfer controller*. Hardware that schedules requests to the transfer controller to move data into and out of VRAM frame memories.

**status bit:** A bit in a status word or register that contains a single piece of status information.

**sync:** A synchronization signal that tells the display where to put the picture. See also *horizontal sync* and *vertical sync*.

## T

**triple:** A row in a table consisting of three columns. For example, an RGB triple contains the red, green, and blue values which define a particular color.

## V

**VC:** *Video controller*. The portion of the 'C8x responsible for the video interface.

**vertical blanking:** Bidirectional vertical timing signals that occur once per frame (once per field for interlaced systems) and have a pulse width defined as an integral number of lines (halflines for an interlaced system). Can be used to disable pixel capture and display during vertical retrace. See also *blanking*.

**vertical sync:** A bidirectional vertical timing signal occurring once per frame with a pulse width defined as an integral number of lines (halflines for interlaced mode). The portion of the composite video signal that tells the receiver where the top of the picture is. See also *horizontal sync*.

**video controller:** See *VC*.

**VIP:** *Video interface palette*. See *RAMDAC*.

**VRAM:** *Video random access memory.* A portion of the microprocessor's memory address space reserved for the temporary storage of video data before it is sent to the display monitor. A type of dynamic read access memory that lets the video circuitry serially access the memory bit by bit. VRAM has separate pins for the processor and video circuitry, is used in high-speed video applications, and is easily interfaced to a video display.

**W**

**window:** A defined rectangular area of virtual space on the display.

**X**

**XPT:** *Externally initiated packet transfer.* A packet transfer initiated by an external device through the 'C8x's  $\overline{\text{XPT}}$  [2:0] inputs.

**Y**

**YUV:** A color space standard in which the luminance (Y) and chrominance (U and V) values are separate components.

# Index

---

---

---

#define. *See* individual macro entry

## A

- accessing internal configuration registers in  
    FIFO 2-16
- active area 2-19, 2-31, 2-33
- AD1848. *See* audio codec
- AE1 bits
  - in CLFLAG1/RDFLAG1 register 2-56
  - in ENABLE2 register 2-53
  - in STFLAG1/CNFLAG1 register 2-57
- AE1 event 2-50
- AE2 bits
  - in CLFLAG0/RDFLAG0 register 2-55
  - in ENABLE1 register 2-52
  - in STFLAG0/CNFLAG0 register 2-56
- AE2 event 2-49
- AF1 bits
  - in CLFLAG0/RDFLAG0 register 2-55
  - in ENABLE1 register 2-52
  - in PCISTAT register 2-59, 2-60
  - in STFLAG0/CNFLAG0 register 2-56
- AF1 event 2-49
- AF2 bits
  - in CLFLAG1/RDFLAG1 register 2-56
  - in ENABLE2 register 2-53
  - in PCISTAT register 2-59, 2-60
  - in STFLAG1/CNFLAG1 register 2-57
- AF2 event 2-50, 2-53
- AFIFOCFG register 2-8, 2-12, 2-14 to 2-18
- AFIFOCMD register 2-8, 2-13, 2-14 to 2-18, 3-15
- AFIFODAT register 2-8, 2-9
- assistance from TI vii
- audcapt.bat file A-11 to A-12
- audcapt.c file A-8 to A-10
- audcapt.lnk file A-10 to A-11
- audio block capture/playback exam-  
    ple A-17 to A-20
- audio buffering 3-5 to 3-8
- audio capture and playback API
  - data types 3-3
  - functions, listing 3-9
  - macros 3-2 to 3-3
- audio codec
  - data formats 2-13
  - example of PIO mode A-21
  - in components diagram 1-3
  - sample rates supported 2-13
- audio codec error event 0 2-50
- audio codec error event 1 2-50
- audio codec event 0 2-49
- audio codec event 1 2-49
- audio codec registers
  - index address (CDCIDX) 2-8, 2-10
  - index data (CDCDAT) 2-8, 2-10
  - programmed input/output (CDCPIO) 2-8, 2-12
  - status (CDCSTAT) 2-8, 2-11 to 2-12
- audio DMA capture example A-8 to A-11
- audio DMA playback example A-12 to A-16
- audio FIFO
  - accessing internal configuration registers 2-16
  - and the audio buffering structure 3-8
  - block diagram 2-9
  - commands supported 2-15
  - description 2-14 to 2-18
  - example of reset code 2-15
  - flag pin configurations 2-17
  - flag truth table 2-18
  - in audio block diagram 2-7
  - internal configuration registers 2-16
- audio FIFO registers
  - command/status (AFIFOCMD) 2-8, 2-13,  
    2-14 to 2-18

- audio FIFO registers (continued)
  - configuration (AFIFOCFG) 2-8, 2-12, 2-14 to 2-18
  - data (AFIFODAT) 2-8, 2-9
- audio FIFO registers, command/status (AFIFOCMD) 3-15
- audio hardware
  - block diagram 2-7
  - description 2-7
  - features 2-2 to 2-4
  - FIFO. *See* audio FIFO
  - registers summary 2-8
- audio programmed I/O (PIO) example A-21 to A-23
- audio.h file 3-2
- AUDIO\_ALAW8 macro 3-2
- AUDIO\_AUX1 macro 3-2
- AUDIO\_CAPTURE macro 3-2
- Audio\_CaptureToMemory function 3-10 to 3-11, A-17
- Audio\_CodecStat function 3-12
- Audio\_Disable function 3-13
- Audio\_Enable function 3-14
- Audio\_FifoStat function 3-15
- Audio\_FillBufs function 3-16
- Audio\_GetCaptureBufs function 3-7, 3-17
- Audio\_GetCodecRegs function 3-18
- Audio\_GetFifoRegs function 3-19
- Audio\_GetMetrics function 3-4, 3-20
- Audio\_GetPlaybackBufs function 3-7, 3-21
- Audio\_Init function 3-22
- Audio\_Install function
  - description 3-23 to 3-24
  - macros used by 3-2
- Audio\_InstallSema function 3-25
- AUDIO\_LINE macro 3-2
- AUDIO\_MET data type 3-3, 3-4, 3-10, 3-20, 3-29
- AUDIO\_MIXED macro 3-2
- AUDIO\_MONO macro 3-2
- AUDIO\_MUTE macro 3-3
- AUDIO\_PCM16 macro 3-2
- AUDIO\_PCM8 macro 3-2
- AUDIO\_PIO macro 3-2
- Audio\_PioIn function 3-26
- Audio\_PioOut function 3-27

- Audio\_PioTest function 3-28, A-21
- AUDIO\_PLAYBACK macro 3-2
- Audio\_PlaybackFromMemory function 3-29 to 3-30, A-17
- Audio\_ProgramAux1 function
  - description 3-31
  - macros used by 3-3
- Audio\_ProgramDacs function
  - description 3-32
  - macros used by 3-3
- Audio\_ProgramDigitalMix function 3-33
- Audio\_ProgramInputs function
  - description 3-34
  - macros used by 3-2
- AUDIO\_PTR data type 3-3 to 3-4, 3-10, 3-17, 3-21, 3-29
- Audio\_SetBufferIndexes function 3-35
- Audio\_SetSampleRate function 3-36
- AUDIO\_STEREO macro 3-2
- AUDIO\_ULAW8 macro 3-2
- Audio\_UnInstall function 3-5, 3-37
- AUDIO\_UNMUTE macro 3-3
- audplay.bat file A-15 to A-17
- audplay.c file A-12
- audplay.lnk file A-15
- audtest.bat file A-20 to A-21
- audtest.c file A-17 to A-19
- audtest.lnk file A-19 to A-20
- autoconfiguration 2-64

## B

- b0 bit 2-70
- BDIS bit 2-59, 2-61
- big-endian mode 2-2, 2-4, 2-45
- bit notation *iv*
- block diagram, system 2-3
- block transfer (blt)
  - read event 2-49
  - read event trigger bit (BLR) 2-61
  - write event 2-49
  - write event trigger bit (BLW) 2-61
- BLR bit 2-59, 2-61
- BLW bit 2-59, 2-61
- BMAH bits 2-68
- BMAH register 2-68, 2-69, 2-70

- BMAL bits 2-68
  - BMAL register 2-68, 2-69, 2-70
  - BMCTRL register 2-68, 2-69, 2-70
  - BMEN bit 2-68, 2-70
  - BMI bits
    - in CLFLAG1/RDFLAG1 register 2-56
    - in ENABLE2 register 2-53
    - in STFLAG1/CNFLAG1 register 2-57
  - BMI event 2-50
  - bootserv.bin file 6-10
  - bootstrapping 2-70 to 2-71, 6-10
  - BRD bits
    - in CLFLAG0/RDFLAG0 register 2-55
    - in ENABLE1 register 2-52
    - in STFLAG0/CNFLAG0 register 2-56
  - BRD event 2-49, 2-52, 2-61
  - buffering
    - audio 3-5
    - video capture 5-5 to 5-7
  - buffering queue structure 3-6
  - BUFFS data type 3-5, 3-6, 3-7
  - burst disable bit (BDIS) 2-61
  - bus controller 2-44
  - bus master address high register (BMAH) 2-69
  - bus master address low register (BMAL) 2-69
  - bus master control register (BMCTRL) 2-69, 2-70
  - bus master event 2-50
  - bus mastering 2-68 to 2-70
  - bus mastering enable bit (BMEN) 2-70
  - BWR bits
    - in CLFLAG0/RDFLAG0 register 2-55
    - in ENABLE1 register 2-52
    - in STFLAG0/CNFLAG0 register 2-56
  - BWR event 2-49, 2-52, 2-61
- C**
- C source debugger 1-2, 1-4, A-8, A-12, A-30
  - 'C80 bus
    - and the memory controller 2-45
    - and video capture FIFO 2-35
    - description 2-5
    - in audio block diagram 2-7
    - in PCI interface block diagram 2-58
    - in video capture block diagram 2-36
    - in video display block diagram 2-20
  - 'C8x DSP device 1-2, 2-2, 2-6
  - 'C8x SDB components 1-2 to 1-3
  - CAP bits
    - example of setting 2-52
    - in CLFLAG0/RDFLAG0 register 2-55
    - in ENABLE0 register 2-52
    - in STFLAG0/CNFLAG0 register 2-56
  - CAP event 2-35, 2-41, 2-49
  - CAPID register 2-37, 2-40, 5-9
  - CAPRST register 2-37, 2-43
  - capptest.bat file A-29 to A-30
  - capptest.c file A-24 to A-28
  - capptest.lnk file A-28 to A-29
  - capture card ID register (CAPID) 2-37, 2-40, 5-9
  - capture FIFO empty flag
    - enable flag bit 2-41
    - interrupt flag bit 2-40
    - interrupt source bit 2-41
  - capture FIFO full flag
    - enable flag bit 2-41
    - interrupt flag bit 2-40
    - interrupt source bit 2-41
  - capture left/right sample bit (CL/R) 2-11
  - capture ready bit (CRDY) 2-11
  - capture upper/lower bit (CU/L) 2-11
  - capture.h file 5-2
  - CAPTURE\_512x512 macro 5-2
  - CAPTURE\_640x480 macro 5-2
  - Capture\_CardPresent function 5-9
  - CAPTURE\_CIF macro 5-2
  - CAPTURE\_CIFK macro 5-2
  - Capture\_Disable function 5-10
  - Capture\_Enable function 5-11
  - Capture\_FillBufs function 5-4, 5-12
  - Capture\_FreeBuffer function 5-5, 5-6, 5-7, 5-13
  - Capture\_GetBuffer function 5-5, 5-6, 5-7, 5-14
  - Capture\_GetDecoderRegs function 5-15
  - Capture\_GetMetrics function 5-4, 5-16
  - Capture\_GetScalerRegs function 5-17
  - Capture\_Init function 5-18
  - Capture\_Install function
    - description 5-19 to 5-20
    - macros used by 5-2, 5-3
  - Capture\_InstallSema function 5-21
  - CAPTURE\_MET data type 5-3, 5-4, 5-16



- CAPTURE\_MONO8 macro 5-3
- CAPTURE\_NTSC macro 5-2
- CAPTURE\_PAL macro 5-2
- CAPTURE\_QCIF macro 5-2
- CAPTURE\_RGB555 macro 5-3
- CAPTURE\_RGB888 macro 5-3
- Capture\_SetInputSource function
  - description 5-22
  - macros used by 5-2
- Capture\_SetScaling function 5-23
- CAPTURE\_SQCIF macro 5-2
- CAPTURE\_SVHS macro 5-2
- Capture\_UnInstall function 5-24
- CAPTURE\_VID1 macro 5-2
- CAPTURE\_VID2 macro 5-2
- CAPTURE\_YUV422 macro 5-3
- category-1 events
  - definition 2-49
  - enable bits 2-52
  - listing 2-49
- category-2 events
  - definition 2-49
  - enable bits 2-53
  - setting/clearing example 2-53
  - listing 2-50
- caution, damage to monitor from unsupported resolutions 4-9, 4-26
- CD0 bits
  - in CLFLAG0/RDFLAG0 register 2-55
  - in ENABLE0 register 2-52
  - in STFLAG0/CNFLAG0 register 2-56
- CD0 event 2-14, 2-49, 2-55
- CD1 bits
  - in CLFLAG0/RDFLAG0 register 2-55
  - in ENABLE0 register 2-52
  - in STFLAG0/CNFLAG0 register 2-56
- CD1 event 2-14, 2-49
- CDCDAT register 2-8, 2-10
- CDCIDX register 2-8, 2-10
- CDCPIO register 2-12
- CDCPIO register 2-8
- CDCSTAT register 2-8, 2-11 to 2-12, 3-12
- CE0 bits
  - in CLFLAG1/RDFLAG1 register 2-56
  - in ENABLE2 register 2-53
  - in STFLAG1/CNFLAG1 register 2-57
- CE0 event 2-14, 2-48, 2-49, 2-50
- CE1 bits
  - in CLFLAG1/RDFLAG1 register 2-56
  - in ENABLE2 register 2-53
  - in STFLAG1/CNFLAG1 register 2-57
- CE1 event 2-14, 2-50, 2-53
- CFIFORST register 2-37
- CL/R bit 2-11
- clear flag register 0 (CLFLAG0) 2-51, 2-55 to 2-56
- clear flag register 1 (CLFLAG1) 2-51, 2-56
- client library (host library) 6-1
- client to server protocol 6-8
- client/server command flow diagram 6-9
- CLIENT\_AF1 macro 6-3
- CLIENT\_AF2 macro 6-3
- CLIENT\_BDIS macro 6-3
- CLIENT\_BINFILE macro 6-2
- CLIENT\_BITSET macro 6-2
- CLIENT\_BLR macro 6-3
- CLIENT\_BLW macro 6-3
- Client\_Boot function
  - and bootstrapping 6-10
  - description 6-12
  - macros used by 6-4
- CLIENT\_BOOTFILE macro 6-2
- CLIENT\_BYTESWAP macro 6-4
- CLIENT\_BYTEWORDSWAP macro 6-4
- Client\_ClearConfigBit function
  - description 6-13
  - macros used by 6-2, 6-3
- Client\_Close function
  - description 6-14
  - macros used by 6-2
- CLIENT\_CLOSED macro 6-2
- CLIENT\_CMNDSIZE macro 6-4
- CLIENT\_COFFFILE macro 6-2
- CLIENT\_DEAD macro 6-2
- CLIENT\_EF1 macro 6-3
- CLIENT\_EF2 macro 6-3
- CLIENT\_EMURST macro 6-4
- CLIENT\_EPOINT macro 6-4
- CLIENT\_FIFOALMOSTFULL macro 6-2
- CLIENT\_FIFOEMPTY macro 6-2
- CLIENT\_FOFF0 macro 6-2
- CLIENT\_FOFF1 macro 6-2
- CLIENT\_FRST macro 6-2

- CLIENT\_FSW0 macro 6-2
- CLIENT\_FSW1 macro 6-2
- CLIENT\_GPIO macro 6-3
- CLIENT\_GPI1 macro 6-3
- CLIENT\_GPO0 macro 6-3
- CLIENT\_GPO1 macro 6-3
- CLIENT\_IAEN macro 6-2
- Client\_Init function 6-15
- Client\_IssueCmnd function
  - description 6-16
  - macros used by 6-2
- CLIENT\_LOAD macro 6-4
- CLIENT\_MAILBOXEMPTY macro 6-2
- CLIENT\_MAILBOXFULL macro 6-2
- CLIENT\_MB1 macro 6-3
- CLIENT\_MB2 macro 6-3
- CLIENT\_MRST macro 6-2
- CLIENT\_NOSWAP macro 6-4
- Client\_Open function
  - description 6-17
  - macros used by 6-2
- CLIENT\_PRGD macro 6-3
- CLIENT\_R0 macro 6-3
- CLIENT\_R1 macro 6-3
- CLIENT\_R10 macro 6-3
- CLIENT\_R2 macro 6-3
- CLIENT\_R3 macro 6-3
- CLIENT\_R4 macro 6-3
- CLIENT\_R5 macro 6-3
- CLIENT\_R6 macro 6-3
- CLIENT\_R7 macro 6-3
- CLIENT\_R8 macro 6-3
- CLIENT\_R9 macro 6-3
- Client\_ReadConfig function
  - description 6-18
  - macros used by 6-2
- Client\_ReadConfigBit function
  - description 6-19
  - macros used by 6-2, 6-3
- Client\_ReadDataFifo function
  - description 6-20
  - macros used by 6-2
- Client\_ReadIo function
  - description 6-21
  - macros used by 6-2
- Client\_ReadMailbox function
  - description 6-22
  - macros used by 6-2
- Client\_Reset function
  - and bootstrapping 6-10
  - description 6-23
  - macros used by 6-2, 6-4
- CLIENT\_RUN macro 6-4
- Client\_SetConfigBit function
  - description 6-24
  - macros used by 6-2, 6-3
- Client\_SetSwapping function
  - description 6-25
  - macros used by 6-2, 6-4
- CLIENT\_STATOK macro 6-2
- Client\_Sync function
  - description 6-26
  - macros used by 6-2
- CLIENT\_TBCOFF macro 6-4
- CLIENT\_TIMEOUT macro 6-2
- CLIENT\_VERBOSE macro 6-4
- CLIENT\_WORDSWAP macro 6-4
- Client\_WriteConfig function
  - description 6-27
  - macros used by 6-2
- Client\_WriteDataFifo function
  - description 6-28
  - macros used by 6-2
- Client\_Writel0 function
  - description 6-29
  - macros used by 6-2
- Client\_WriteMailbox function
  - description 6-30
  - macros used by 6-2
- CNFLAG0 register 2-51, 2-56
- CNFLAG1 register 2-51, 2-57
- codec index address register (CDCIDX) 2-8, 2-10
- codec index data register (CDCDAT) 2-8, 2-10
- codec PIO data register (CDCPIO) 2-8, 2-12
- codec status register (CDCSTAT) 2-8, 2-11 to 2-12, 3-12
- color modes
  - defined 4-5
  - direct 4-2, 4-5
  - pseudocolor 4-2, 4-5
  - true 4-2, 4-5
- color palette RAM 2-22, 2-23

composite video 1-2, 2-38, 2-39  
*See also* CVBS video  
 condition flag register 0 (CNFLAG0) 2-51, 2-56  
 condition flag register 1 (CNFLAG1) 2-51, 2-57  
 CRDY bit 2-11  
 CU/L bit 2-11  
 CVBS video 1-2, 1-4, 2-4, 2-35  
 CVBS1 mode 2-38, 2-39  
 CVBS1 register 2-37, 2-38  
 CVBS2 mode 2-39  
 CVBS2 register 2-37, 2-38, 2-39

## D

damage to monitor from unsupported resolutions 4-9, 4-26  
 data buses 2-5  
 data types  
   audio capture and playback API 3-3  
   host communications API 6-5  
   video capture API 5-3  
   video display API 4-4  
 DATCLK bit 2-30  
 DESC. *See* SAA7196 video decoder/scaler  
 device pin notation iv  
 direct color mode 4-2, 4-5  
 direct memory access (DMA) mode 2-7  
 display control registers  
   DIS0CON register 2-21, 2-25  
   DIS1CON register 2-21, 2-28, 2-30  
   DIS2CON register 2-21, 2-29  
 display EPLD block diagram 2-27  
 display I/O 2-20, 2-34  
 display timing parameters 2-33, 4-5  
 display window 2-33  
 display.h file 4-2  
 DISPLAY\_ACTIVE macro 4-3  
 DISPLAY\_BUFF1 macro 4-3  
 DISPLAY\_BUFF2 macro 4-3  
 DISPLAY\_D444 macro 4-2  
 DISPLAY\_D555 macro 4-2  
 DISPLAY\_D565 macro 4-2  
 DISPLAY\_D664 macro 4-2  
 DISPLAY\_DBGRX macro 4-2  
 Display\_Disable function 4-11

DISPLAY\_DXRGB macro 4-2  
 Display\_Enable function 4-12  
 Display\_FillBufs function 4-13  
 Display\_GetBuffer function  
   description 4-14  
   macros used by 4-3  
 Display\_GetMetrics function 4-8, 4-15  
 Display\_GetTvpRegs function 4-16  
 DISPLAY\_INACTIVE macro 4-3  
 Display\_Init function 4-17  
 Display\_InstallSema function 4-18  
 Display\_InstallTimingTable function 4-7, 4-8, 4-19  
 DISPLAY\_MET data type 4-4, 4-8, 4-15  
 Display\_MoveWindow function 4-20  
 DISPLAY\_MT data type 4-4, 4-5 to 4-8, 4-19  
 DISPLAY\_OVERLAY macro 4-3  
 DISPLAY\_P8 macro 4-2  
 DISPLAY\_PASSTHROUGH macro 4-3  
 Display\_ReadPalette register 4-21  
 Display\_SetBufferAddresses function 4-22  
 Display\_SetDotClock function 4-23  
 Display\_SetMode function  
   description 4-25 to 4-26  
   macros used by 4-2, 4-3  
 Display\_SetOverlayParams function 4-8, 4-27 to 4-28  
 Display\_SetPaletteAddress function, 4-29  
 Display\_SetPitch function 4-30  
 Display\_SetPixel function  
   description 4-31  
   macros used by 4-3  
 Display\_SetSyncPolarities function 4-32  
 Display\_SetVgaPalette function 4-33  
 Display\_SetWindow function 4-34  
 DISPLAY\_T444 macro 4-2  
 DISPLAY\_T555 macro 4-2  
 DISPLAY\_T565 macro 4-2  
 DISPLAY\_T664 macro 4-2  
 DISPLAY\_TBGRX macro 4-2  
 Display\_ToggleBuffers function 4-35  
 Display\_TvpRegOut function 4-37  
 DISPLAY\_TXRGB macro 4-2  
 DISPLAY\_VIDEO macro 4-3  
 Display\_WaitEndOfFrame function 4-38  
 Display\_WritePalette function 4-39

DisplaySetGreyScalePalette function 4-24  
 disptest.bat file A-39 to A-40  
 disptest.c file A-30 to A-38  
 disptest.lnk file A-38 to A-39  
 DMA (direct memory access) mode 2-7  
 dot 2-19, 2-31  
 dot clock 2-32, 2-33  
 dot clock frequency ( $F_d$ ). *See* dot rate  
 dot rate 2-19, 2-31, 2-32  
 DRAM  
     and memory controller 2-45  
     features 2-4  
     in components diagram 1-3  
 dynamic random-access memory. *See* DRAM

## E

e0–e7 bits 2-41  
 EF1 bits  
     in CLFLAG1/RDFLAG1 register 2-56  
     in ENABLE2 register 2-53  
     in PCISTAT register 2-59, 2-60  
     in STFLAG1/CNFLAG1 register 2-57  
 EF1 event 2-50  
 EF2 bits  
     in CLFLAG0/RDFLAG0 register 2-55  
     in ENABLE1 register 2-52  
     in PCISTAT register 2-59, 2-60  
     in STFLAG0/CNFLAG0 register 2-56  
 EF2 event 2-49  
 EINT bits 2-54  
 emurst.exe file 6-4, 6-23  
 ENABLE0 register 2-51, 2-52  
 ENABLE1 register 2-51, 2-52  
 ENABLE2 register 2-51, 2-53  
 event  
     category-1 table 2-49  
     category-2 table 2-50  
     clearing, example 2-56  
     definition 2-48  
     setting/clearing category-2, example 2-53  
     triggering 2-48  
 event destination 2-48  
 event enable register 0 (ENABLE0) 2-51, 2-52  
 event enable register 1 (ENABLE1) 2-51, 2-52  
 event enable register 2 (ENABLE2) 2-51, 2-53

event signal 2-48  
 event source 2-48  
 event state register (EVSTATE) 2-51, 2-54  
 event trigger 2-61  
     definition 2-48  
     example 2-42  
 even-to-odd field transition enable flag bit 2-41  
 even-to-odd field transition interrupt flag bit 2-40  
 even-to-odd field transition interrupt source bit 2-41  
 EVSTATE register 2-51, 2-54

## F

$F_d$  (dot clock frequency). *See* dot rate  
 FF1 bits  
     in CLFLAG1/RDFLAG1 register 2-56  
     in ENABLE2 register 2-53  
     in STFLAG1/CNFLAG1 register 2-57  
 FF1 event 2-50  
 FF2 bits  
     in CLFLAG1/RDFLAG1 register 2-56  
     in ENABLE2 register 2-53  
     in STFLAG1/CNFLAG1 register 2-57  
 FF2 event 2-50  
 FIFO boot location 2-68  
 FIFO data location 2-68  
 FIFO flag pin configurations 2-17  
 FIFO flag truth table 2-18  
 FIFO header location 2-67  
 FIFO mailbox1 full flag bit (MB1) 2-60  
 FIFO mailbox2 full flag bit (MB2) 2-59  
 FIFO swap setting bits (FSW) 2-62  
 FIFO1 almost empty flag event 2-50  
 FIFO1 almost full flag bit (AF1) 2-60  
 FIFO1 almost full flag event 2-49  
 FIFO1 empty flag bit (EF1) 2-60  
 FIFO1 empty flag event 2-50  
 FIFO1 full flag event 2-50  
 FIFO2 almost empty flag event 2-49  
 FIFO2 almost full event 2-50  
 FIFO2 almost full flag bit (AF2) 2-60  
 FIFO2 empty flag bit (EF2) 2-60  
 FIFO2 empty flag event 2-49  
 FIFO2 full flag event 2-50  
 FOFF bits 2-59, 2-62

frame timer registers 4-7

FRM bits

in CLFLAG0/RDFLAG0 register 2-55

in ENABLE0 register 2-52

in STFLAG0/CNFLAG0 register 2-56

FRM event 2-35, 2-49

FRST bit 2-59, 2-62

FSW bits 2-59, 2-62

functions, listing

audio capture and playback API 3-9

host communications API 6-11

video capture API 5-8

video display API 4-10

$F_v$  (vertical frequency). *See* refresh rate

## G

general-purpose input bits (GPI) 2-59

general-purpose output bits (GPO) 2-61

general-purpose video capture event 2-49

GPI bits 2-59

GPO bits 2-59, 2-61

## H

hardware functions list 1-4

hardware modules 2-2

hclient.h file 6-2

horizontal sync times 4-6

host communications API

data type 6-5

functions, listing 6-11

macros 6-2 to 6-4

host library (client library) 6-1

hsdbdrv.h file B-1, B-4

hsdbdrv.lib file 6-2

HSYNC0 falling edge enable flag bit 2-41

HSYNC0 falling edge interrupt flag bit 2-40

HSYNC0 falling edge interrupt source bit 2-41

HSYNC1 invert bit (IHS1) 2-29

## I

I/O bus

and IDT72520 FIFO 2-14

and the memory controller 2-46

audio registers accessible by 2-8

bus mastering registers accessible by 2-69

description 2-5

in audio block diagram 2-7

in host address space 2-67

in PCI interface block diagram 2-58

in video capture block diagram 2-36

in video display block diagram 2-20

interrupt controller registers accessible by 2-51

video capture registers accessible by 2-37

video display registers accessible by 2-21

I/O space, host 2-67

i0–i7 bits 2-40

I2CCTRL bits 2-42

I2CCTRL register 2-37, 2-42, 2-44

I2CDAT bits 2-42

I2CDAT register 2-37, 2-42, 2-44

I2CRST register 2-37, 2-43

I2CSTAT bits 2-43

I2CSTAT register 2-37, 2-43, 2-44

IAEN bit 2-59, 2-61

ICE bit 2-25, 2-26

ICS1574. *See* pixel clock generator

ICS1574 clock bit (ICSC) 2-28

ICS1574 data bit (ICSD) 2-28, 2-30

ICS1574 hold bit (ICSH) 2-28

ICSC bit 2-28, 2-30

ICSD bit 2-28, 2-30

ICSH bit 2-28, 2-30

IDIS bit 2-68, 2-70

IDT72520. *See* audio FIFO

IEEE 1149.1 emulation support 1-3

IEEE 1149.1 standard 1-3

IHS bit 2-25

IHS1 bit 2-29

increment disable bit (IDIS) 2-70

index address bits (IXA) 2-10

individual buffer structure 3-8

individual queue buffer structure 3-7

initialization bit (INIT) 2-10

INT bit 2-10, 2-11, 2-12

INTEN register 2-37, 2-41, 2-42  
 interaction between client and server APIs 6-6  
 interrupt controller (memory/interrupt controller)  
   *See also* memory controller  
   and bootstrapping 2-70  
   and bus-matching FIFO 2-14, 2-17  
   description 2-48 to 2-57  
   events from video capture 2-35  
   features 2-4  
   in audio block diagram 2-7  
   in PCI interface block diagram 2-58  
   in video capture block diagram 2-36  
   registers summary 2-51  
 interrupt enable register (INTEN) 2-37, 2-41  
 interrupt flag register (INTREG) 2-37, 2-40  
 interrupt source register (INTSRC) 2-37,  
   2-41 to 2-42  
 interrupt status sticky bit (INT) 2-10, 2-12  
 INTREG register 2-37, 2-40, 2-42  
 INTSRC bits 2-41  
 INTSRC register 2-37, 2-41 to 2-42  
 invert horizontal sync bit (IHS) 2-25  
 invert pixel clock enable bit (ICE) 2-26  
 invert vertical sync bit (IVS) 2-25  
 ISRC bits 2-38  
 ISRC register 2-37, 2-38  
 IVS bit 2-25  
 IVS1 bit 2-29  
 IXA bits 2-10

## K

kit contents 1-2

## L

LINT bit 2-54

## M

macros

  audio capture and playback API 3-2 to 3-3  
   host communications API 6-2 to 6-4  
   video capture API 5-2 to 5-3  
   video display API 4-2 to 4-3

MAIL location 2-67  
 MASKEN0 register 6-29  
 master processor (MP) 1-2  
 master reset bit (MRST) 2-62, 2-70  
 MB1 bit 2-59, 2-60, 2-67  
 MB2 bit 2-59, 2-67  
 MCE bit 2-10  
 MCI bits  
   in CLFLAG1/RDFLAG1 register 2-56  
   in ENABLE2 register 2-53  
   in STFLAG1/CNFLAG1 register 2-57  
 MCI event 2-50  
 memory  
   DRAM 2-4, 2-45  
   RAM 2-6, 2-22, 2-23  
   SAM 2-46  
   SRAM 2-6  
   VRAM  
     *and memory controller* 2-46  
     *and pixel resolution* 2-31 to 2-33  
     *features* 2-4  
     *in video display hardware* 2-19 to 2-20  
 memory controller (memory/interrupt controller)  
   *See also* interrupt controller  
   description 2-45 to 2-47  
   features 2-4  
   in audio block diagram 2-7  
   in components diagram 1-3  
   in PCI interface block diagram 2-58  
 memory controller event 2-50  
 MEN bit 2-25  
 metric parameters  
   display 4-8  
   video capture 5-4  
 miniboot.bin file 6-10  
 mix enable bit (MEN) 2-25, 2-26  
 mixed mode 2-25, 2-29, 2-33, 4-3  
 mode change enable bit (MCE) 2-10  
 monitor timing parameters 4-4, 4-5  
 mono mode 2-13  
 MRST bit 2-59, 2-62, 2-70

## N

noninterlaced video 4-5  
 notational conventions iv

## NTSC

- definition D-7
- video 1-2, 2-4, 2-35

**O**

- odd-to-even field transition enable flag bit 2-41
- odd-to-even field transition interrupt flag bit 2-40
- odd-to-even field transition interrupt source bit 2-41
- opcode bits 2-13, 2-14, 2-15, 2-16
- operand bits 2-13, 2-14, 2-15, 2-16
- overlay mode 4-3, 4-25, 4-27
  - See *also* mixed mode
- overlay, video 2-25, 2-33 to 2-34, 4-3, 4-8, 4-25, 4-27

**P**

- packet transfer 2-7, 2-9, 2-35, 2-46, 2-54, A-8
  - See *also* PDPT (peripheral data packet transfer)
- page mode 2-45, 2-46
- PAL
  - definition D-8
  - video 1-2, 2-4, 2-35
- PAL86 bit 2-28, 2-30
- PALADDR register 2-21, 2-22, 2-24
- PALADWR register 2-21, 2-22
- palette graphics output 2-25, 2-26, 2-33
- palette holding register (PALHOLD) 2-21, 2-22, 2-24
- palette read address register (PALADDR) 2-21, 2-22, 2-24
- palette write address register (PALADWR) 2-21, 2-22
- PALHOLD register 2-21, 2-22, 2-24
- parallel processor (PP) 1-2
- PCF8584 I<sup>2</sup>C control register (I2CCTRL) 2-37, 2-42
- PCF8584 I<sup>2</sup>C controller chip 2-36, 2-42, 2-43, 2-44
- PCF8584 I<sup>2</sup>C data register (I2CDAT) 2-37, 2-42
- PCF8584 I<sup>2</sup>C status register (I2CSTAT) 2-37, 2-43

## PCI bits

- in CLFLAG0/RDFLAG0 register 2-55
- in ENABLE0 register 2-52
- in STFLAG0/CNFLAG0 register 2-56

## PCI bus

- and bus master writes 2-68
- and the memory controller 2-47
- description 2-5
- in PCI interface block diagram 2-58

## PCI bus mastering, registers summary 2-69

## PCI event 2-49, 2-61

## PCI FIFO

- access by 'C80 2-63 to 2-64
- block diagram 2-63
- description 2-58, 2-62 to 2-63
- host-to-SDB (FIFO1) 2-49, 2-50, 2-60, 2-63, 2-64, 2-67, 2-68
- in host address space 2-66 to 2-68
- mailbox
  - host-to-SDB (mailbox1)* 2-60, 2-63, 2-64, 2-67
  - SDB-to-host (mailbox2)* 2-59, 2-63, 2-64, 2-67
- mailboxes 2-62, 2-64, 2-67
- resetting 2-62
- SDB-to-host (FIFO2) 2-49, 2-50, 2-60, 2-63, 2-64, 2-68, 2-70

## PCI FIFO offset bit (FOFF) 2-62

## PCI FIFO reset bit (FRST) 2-62

## PCI interface

- See *also* PCI FIFO
- block diagram 2-58
- description 2-58 to 2-71
- in components diagram 1-3
- registers
  - bus mastering, listing* 2-69
  - PCI status register (PCISTAT)* 2-59

## PCI interrupt A enable bit (IAEN) 2-61

## PCI status register (PCISTAT)

- and bootstrapping 2-70 to 2-71, 6-10
- and Client\_ClearConfigBit function 6-13
- and Client\_ReadConfig function 6-18
- and Client\_ReadConfigBit function 6-19
- and Client\_ReadIo function 6-21
- and Client\_ReadMailbox function 6-22
- and Client\_SetConfigBit function 6-24
- and Client\_WriteConfig function 6-27
- and host reads 2-67, 2-68
- bits and fields 2-59 to 2-62

PCI status register (PCISTAT) (continued)  
     description 2-59 to 2-62  
     location in host address space 2-67  
 PDPT (peripheral data packet transfer) 2-35, 2-45, 2-46  
     *See also* package transfer  
 PELMASK register 2-21, 2-23  
 peripheral data packet transfer (PDPT) 2-35, 2-45, 2-46  
     *See also* packet transfer  
 peripheral driver libraries 1-4  
 PGD bits  
     in CLFLAG1/RDFLAG1 register 2-56  
     in ENABLE2 register 2-53  
     in STFLAG1/CNFLAG1 register 2-57  
 PGD event 2-50  
 pin notation *iv*  
 PIO (programmed input/output) mode 2-7, 2-12, A-21  
 piotest.bat file A-23 to A-24  
 piotest.c file A-21 to A-22  
 piotest.lnk file A-22 to A-23  
 pixel  
     defined 2-19  
     resolution 2-19, 2-31 to 2-33  
 pixel clock frequency 2-19, 2-33  
     *See also* dot rate  
 pixel clock generator  
     and DIS1CON register 2-28  
     description 2-30 to 2-31  
     in video display block diagram 2-20  
     sample program 2-30  
 pixel depth 2-19, 2-31, 2-32  
 pixel read-mask register (PELMASK) 2-21, 2-23  
 pixel resolution 1-3, 1-4  
 playback left/right sample bit (PL/R) 2-11  
 playback ready bit (PRDY) 2-12  
 playback upper/lower bit (PU/L) 2-11  
 plug and play 2-64 to 2-65  
 plug-in card 1-2  
 porch times 4-6  
 PRDY bit 2-11, 2-12  
 PRGD bit 2-59  
 programmed input/output (PIO) mode 2-7, 2-12, A-21  
 programming done bit (PRGD) 2-59

programming done event 2-50  
 progressive scan (noninterlaced video) 4-5  
 protocol, client to server 6-8  
 pseudocolor mode 2-23, 4-2, 4-5  
 PU/L bit 2-11

## R

r1 bit 2-70  
 RAM 2-6, 2-22, 2-23  
 read flag register 0 (RDFLAG0) 2-51, 2-55 to 2-56  
 read flag register 1 (RDFLAG1) 2-51, 2-56  
 read input source setting register (ISRC) 2-37, 2-38  
 reduced instruction set computer (RISC) 1-2  
 refresh rate 1-3, 2-19, 2-31, 2-32  
 register bit notation *iv*  
 registers  
     audio hardware, listing 2-8  
     interrupt controller, listing 2-51  
     PCI bus mastering, listing 2-69  
     PCI interface 2-59  
     video capture hardware, listing 2-37  
     video display hardware, listing 2-21  
 related documentation *v to vi*  
 reset PCF8584 I<sup>2</sup>C controller register (I2CRST) 2-37, 2-43  
 reset video capture EPLD register (CAPRST) 2-37, 2-43  
 reset video capture FIFO register (CFI-FORST) 2-37  
 ROW bits  
     in CLFLAG0/RDFLAG0 register 2-55  
     in ENABLE0 register 2-52  
     in STFLAG0/CNFLAG0 register 2-56  
 ROW event 2-35, 2-48, 2-49

## S

SAA7196 video decoder/scaler (DESC) 2-36, 2-44  
 sample overrun/underrun bit (SOUR) 2-11  
 samples directory A-1  
 scaling resolutions supported, video capture 5-4  
 scaling example, video capture A-24 to A-29  
 sdbboot.out file 6-10  
 sdbdrvs.h file B-1, B-2



- sdbdrvs.lib 3-2
- sdbdrvs.lib file 4-2, 5-2, 6-2
- sdbrst.exe file 6-10
- SEN bit 2-28
- SENSE from TVP3020 bit (SEN) 2-28
- serial access memory. *See* SAM
- serial access memory (SAM) 2-46
- server library (SDB library) 6-1
- SERVER\_CMND\_SIZE macro 6-4
- Server\_Done function 6-31
- Server\_GetCmd function 6-32
- Server\_Init function 6-33
- Server\_Install function 6-34
- Server\_InstallSema function 6-35
- Server\_ReadDataFifo function 6-36
- Server\_Sync function 6-37
- Server\_UnInstall function 6-38
- Server\_WriteDataFifo function 6-39
- set flag register 0 (STFLAG0) 2-51, 2-56
- set flag register 1 (STFLAG1) 2-51, 2-57
- set input to CVBS1 register (CVBS1) 2-37, 2-38
- set input to CVBS2 register (CVBS2) 2-37, 2-39
- set input to S-VHS register (SVHS) 2-37, 2-38
- software development board (SDB)
  - description 1-1
  - hardware
    - audio 2-2, 2-7 to 2-18
    - block diagram 2-3
    - data buses 2-5
    - DRAM 2-4, 2-45
    - functions 1-4
    - interrupt controller 2-4, 2-48 to 2-57
    - memory controller 2-4, 2-45 to 2-47
    - overview 2-2 to 2-5
    - PCI interface 2-4, 2-58 to 2-71
    - TMS320C80 2-2, 2-6, 2-45, 2-46
    - video capture 2-4, 2-35 to 2-44
    - video display 2-2, 2-19 to 2-34
    - VRAM 2-4, 2-31 to 2-33, 2-46
  - illustration 1-3
  - kit, contents 1-2
  - memory address assigned 2-64
  - PCI plug-in card, components 1-2
  - peripheral driver libraries 1-4
  - plug and play 2-64
- SOUR bit 2-11
- SRAM 2-6
- ssserver.h file 6-2
- state of the 'C80  $\overline{\text{EINT}}[3:1]$  pins bits (EINT) 2-54
- state of the 'C80  $\overline{\text{LINT}}4$  pin bit (LINT) 2-54
- state of the 'C80  $\overline{\text{XPT}}[2:0]$  pins bits (XPT) 2-54
- static random-access memory. *See* SRAM
- stereo mode 2-13
- STFLAG0 register 2-51, 2-56
- STFLAG1 register 2-51, 2-57
- S-VHS mode 2-38, 2-39
- S-VHS video 1-2, 1-4, 2-4, 2-35, 2-36, 2-38
- SVHS register 2-37, 2-38
- synchronization of client and server
  - APIs 6-6 to 6-7
- system block diagram 2-3
- system-level overview 2-2 to 2-5

## T

- TC (transfer controller) 2-6
- technical support vii
- test bus controller event 2-50
- TFRAME enable flag bit 2-41
- TFRAME interrupt flag bit 2-40
- TFRAME interrupt source bit 2-41
- TMS320C8x digital signal processor (DSP). *See* 'C8x DSP device
- transfer controller (TC) 2-6, 2-35, 2-46
- transfer request disable bit (TRD) 2-10
- trigger an event 2-61
- triggering an event
  - definition 2-48
  - example 2-42
- TROW enable flag bit 2-41
- TROW interrupt flag bit 2-40
- TROW interrupt source bit 2-41
- true color mode 4-2, 4-5
- TVP3020 data register (TVPDAT) 2-21, 2-24
- TVP3020 index register (TVPIDX) 2-21, 2-24
- TVP3020 palette 8/6 mode bit (PAL86) 2-28
- TVP3020 RAMDAC
  - and the DIS0CON register 2-25, 2-26
  - and the PALADDR register 2-24
  - and the PALADWR register 2-21
  - and the PALHOLD register 2-22

TVP3020 RAMDAC (continued)  
     and the TVPDAT register 2-24  
     and the TVPIDX register 2-24  
     and video overlay (mixed mode) 2-33  
     in display I/O block diagram 2-34  
     in video display block diagram 2-20  
 TVPDAT register 2-21, 2-24  
 TVPIDX register 2-21, 2-24

## V

VC (video controller) 2-6  
 vertical frequency ( $F_v$ ) 2-19  
     *See also* refresh rate  
 VGA bit 2-25, 2-26  
 VGA pass-through mode 2-26, 2-33, 4-3, 4-25  
 VGA/mix enable bit (VGA) 2-26  
 video capture API  
     buffering 5-5 to 5-7  
     data type 5-3  
     functions, listing 5-8  
     macros 5-2 to 5-3  
     metric parameters 5-4  
     scaling resolutions supported 5-4  
 video capture condition 2-35, 2-40  
 video capture FIFO  
     description 2-35, 2-46  
     in video capture block diagram 2-36  
     resetting 2-37  
 video capture frame event 2-49  
 video capture hardware  
     block diagram 2-36  
     description 2-35  
     features 2-4  
     FIFO 2-35  
     registers, listing 2-37  
 video capture row event 2-49  
 video capture scaling example A-24 to A-29  
 video capture-process-display example A-2 to A-7  
 video controller (VC) 2-6  
 video digitizer 1-2, 2-39

video display API  
     data types 4-4  
     functions, listing 4-10  
     macros 4-2 to 4-3  
 video display hardware  
     block diagram 2-20  
     description 2-19  
     features 2-2  
     registers summary 2-21  
 video display modes, overlay (mixed) 2-29  
 video display test example A-30 to A-40  
 video interface palette (VIP) 1-3  
 video mode 4-3, 4-17, 4-25  
 video overlay 2-33 to 2-34, 4-3, 4-8, 4-25, 4-27  
 video random-access memory. *See* VRAM  
 video.bat file A-7 to A-8  
 video.c file A-2 to A-6  
 video.lnk file A-6  
 VRAM  
     and memory controller 2-46  
     and pixel resolution 2-31 to 2-33  
     features 2-4  
     in components diagram 1-3  
     in video display hardware 2-19 to 2-20  
 VSYNC0 falling edge enable flag bit 2-41  
 VSYNC0 falling edge interrupt flag bit 2-40  
 VSYNC0 falling edge interrupt source bit 2-41  
 VSYNC1 invert bit (IVS1) 2-29

## W

window mode bit (WIN) 2-25

## X

XLI bits  
     in CLFLAG1/RDFLAG1 register 2-56  
     in ENABLE2 register 2-53  
     in STFLAG1/CNFLAG1 register 2-57  
 XLI event 2-50  
 XPT bits 2-54