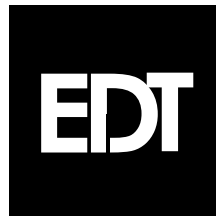


SCD

SBus Configurable DMA Interface

USER'S GUIDE

008-00419-04



The information in this document is subject to change without notice and does not represent a commitment on the part of Engineering Design Team, Inc. The software described in this document is furnished under a license agreement or nondisclosure agreement. The software may be used or copied only in accordance with the terms of the agreement.

No part of this manual may be reproduced or transmitted in any form or by any means, electronic or mechanical, without the express written agreement of Engineering Design Team, Inc.

© Copyright Engineering Design Team, Inc. 1992–1997. All rights reserved.

Refer questions or problems with this manual or the hardware or software described herein to:

Engineering Design Team, Inc.
1100 NW Compton Drive, Suite 306
Beaverton, Oregon 97006

Phone (503) 690-1234
Fax (503) 690-1243
E-mail: info@edt.com
Web: www.edt.com

Sun, SunOS, SBus, SPARC, and SPARCstation are trademarks of Sun Microsystems, Incorporated.

UNIX and OPEN LOOK are registered trademarks of UNIX System Laboratories, Inc.

Xilinx is a registered trademark of Xilinx, Inc.

EDT and Engineering Design Team are trademarks of Engineering Design Team, Inc.

Contents

Overview	1
Installation.....	2
Installing the Hardware.....	2
Installing the Software	2
Using SunOS Version 4.1	2
Using System V Release 4 (Solaris 2.x)	4
Building the Sample Programs	4
Included Files	4
Input and Output	5
Elements of SCD Applications.....	5
DMA Library Routines	6
Error Conditions	13
Hardware Interface Protocol	14
Electrical Interface	14
RS-422	15
Pseudo-ECL	15
LVDS.....	16
Signals	16
Timing	18
Connector Pinout.....	18
Registers.....	20
SBus Addresses	20
Output FIFO Almost Empty Threshold Register.....	20
Output FIFO Almost Full Threshold Register.....	21
Input FIFO Almost Empty Threshold Register	21
Input FIFO Almost Full Threshold Register	21
Current DMA Address Register	22
Next DMA Address Register	22
Current Count Register.....	22
Control and Next Count Register.....	23
Xilinx Programming Register	23
Direction Control Register.....	24
Command Register	25
Data Path Status Register	26
Stat Register	26
Funct Register.....	26
Stat Polarity Register	27
Specifications	28
SBus Compliance	28
Device Data Transfer.....	28
Software	28
Power.....	28
Environmental.....	28
Physical	28
Appendix A ioctl() Parameters.....	29

Figures

Balanced Differential Circuit	14
RS-422 Data Signalling Rate and Cable Length.....	15
PECL Data Signalling Rate and Cable Length	16
SCD Timing	18
SBus Addresses	20

Tables

General DMA Library Routines	6
Error Codes and Conditions	13
Signals	17
Timing Specifications	18
Connector Pinout.....	19
The Output FIFO Almost Empty Threshold	21
The Output FIFO Almost Full Threshold	21
The Input FIFO Almost Empty Threshold.....	21
The Input FIFO Almost Full Threshold	22
The Current DMA Address Register	22
The Next DMA Address Register	22
The Current Count Register	22
The Control and Next Count Register.....	23
Xilinx Programming Register	24
The Direction Control Register	25
The Command Register	25
The Data Path Status Register.....	26
The Stat Register	26
The Funct Register	26
The Stat Polarity Register	27

Overview

The SBus Configurable DMA Interface (SCD) is a single-slot, 16-bit parallel input/output interface for SBus-based computer systems. It is designed for continuous input or output between a user device and SBus host memory. This interface is typically used to move data to or from an SBus host computer to devices such as scanners, plotters, imaging devices, or research prototypes. The SCD uses a simple synchronous protocol for transferring data.

The interface is synchronous—all data and control signals are transmitted with a clock signal. Either the SCD interface or the user device can generate the receive or transmit timing, or each can generate its own transmit timing.

The SCD-20 uses RS-422 signal levels and operates at clock rates of up to 10 MHz or 20 megabytes per second. It uses 1KB input and output FIFOs. The SCD-40 uses AT&T pseudo-emitter-coupled logic (PECL) signal levels and operates at clock rates of up to 20 MHz or 40 megabytes per second. All signals are differential. The SCD-60 uses low-voltage differential signaling and operates at clock rates of up to 30 MHz or 60 megabytes per second. Both the SCD-40 and the SCD-60 use 2 KB input and output FIFOs.

The input and output FIFO buffers smooth data transfer between the SBus and the user device, as well as accommodating data during the transition from one DMA to the next. DMA transfers are queued in hardware, minimizing the amount of FIFO required.

The SCD boards fully support the requirements of the SunOS operating system.

This document explains how to install the SCD interface and driver and how to write applications for it. It is divided into the following sections:

Installation	describes how to install the board and its related software.
Input and Output	describes the programming interface system calls, read(), write() and library calls.
Hardware Interface Protocol	provides a connector pin-out diagram and describes the SCD signals and timing.
Registers	describes the hardware registers.
Specifications	lists the product specifications.
ioctl() Parameters	describes the parameters to the <i>ioctl</i> s.

Installation

Installing the SCD Board interface is a two-step process. First you must physically install the board inside the host computer. Then you must install the software driver so that applications can access the SCD Board. Hardware installation is described in the following section. Software installation is described in the section after.

Installing the Hardware

The SCD Board is a single-slot SBus board. To install it, refer to your SBus host computer documentation. Also see "Installing SBus Cards in Desktop SPARCstations" Sun part no. 800-6635-11 for complete information on installing an SBus board.

Use the following procedure to install the SCD Board:

CAUTION

Both the SCD and your SBus host computer contain static-sensitive components. Install the SCD at a static-free work area. If a static-free work area is not available, take the following precautions to reduce the risk of component damage:

1. Remove from the immediate area all materials that can generate or hold a static charge.
 2. Discharge yourself by touching both hands to a metal portion of the host computer's chassis before you open the host computer or open the SCD static-shielded bag.
-
-

1. Unpack the SCD Board from the shipping packaging. Do not remove the SCD Board from the static shielding bag until you remove all other packaging materials from the area and establish a static-free work area.
2. Install the SCD Board in the SBus host, following the directions provided with the SBus host. The SCD Board can be installed in any DMA slot.

To remove the SCD Board, reverse the installation procedure.

The SCD Board connects to your device with a cable. This cable is typically device-specific.

Installing the Software

The SBus Configurable DMA (SCD) Board can run on a Sun workstation using either SunOS Version 4.1 or Solaris 2.x (System V Release 4, or SVR 4). The installation procedures differ. Both are given below.

Using SunOS Version 4.1

If you are using SunOS Version 4.1, use the following procedure to install the SCD Board driver:

1. Become root or superuser.
2. Change to the directory in which you wish to install the SCD Board driver.
3. Place the diskette that came with the SCD Board into the diskette drive.

4. The SCD Board driver and related files are included on a diskette in *tar* format. To copy them to your hard disk, enter:

```
tar xvf /dev/rfd0
```

5. The *tar* program extracts a number of files. (The list of files distributed is provided in the section entitled **Included Files**.) The SCD Board diskette contains versions of the SCD Board driver for a variety of Sun platforms and versions of the Sun operating system. The installation program installs the correct driver based on the host platform and operating system version.
6. To install the driver, enter:

```
make install
```

The makefile provided installs and loads the SCD Board driver.

7. During the installation, the following question appears on the display:

```
Automatically load the scd driver during each reboot? [y|n] (y):
```

Entering *y* (or simply typing <Return>) causes the SCD board driver to be loaded whenever you reboot your host computer. If you respond with *n*, you must manually reload the driver after rebooting. To do so, enter:

```
make load
```

8. During the installation, the following question appears on the display:

```
How many scd devices do you want? (1):
```

You can install as many SCD boards in your system as you have DMA SBus slots available. Enter the number corresponding to the number of SCD boards you have installed in your system. If you simply type <Return>, one SCD device entry is installed.

NOTE: If you anticipate installing more than one SCD board into your system, install as many SCD board device entries as you will ultimately require. The extra device entries will do no harm and will be there when you need them, saving you a step.

9. If the SCD board has not been installed inside the host computer, or has been installed incorrectly, the following message appears on the display:

```
Can't load this module
```

If you see this message, go back to the section entitled **Installing the Hardware** and reinstall the board.

To unload the SCD board driver:

1. Change to the directory in which you placed the SCD board files, if you are not already there.
2. Become root or superuser.
3. Enter:

```
make unload
```

Using System V Release 4 (Solaris 2.x)

If you are using Sun System V Release 4 (Solaris 2.x), use the following procedure to install the SCD board driver:

1. Become root or superuser.
2. Place the diskette that came with the SCD board into the diskette drive.
3. Enter:

```
volcheck
pkgadd -d /vol/dev/aliases/floppy0 EDTscd
```

To remove the SCD board driver:

1. Become root or superuser.
2. Enter:

```
pkgrm EDTscd
```

For further details, consult your Solaris 2.0 documentation, or call Engineering Design Team, Inc.

Building the Sample Programs

To build any of the example programs, enter the command:

```
make file
```

where *file* is the name of the example program you wish to install.

To build and install all the example programs, simply enter the command:

```
make
```

All example programs display a message that explains their usage when you enter their names without parameters.

Included Files

See the *readme* file for a complete, up-to-date listing of files included on the SCD driver release diskette.

Input and Output

The SCD device driver can perform two kinds of DMA transfers: continuous and noncontinuous. For noncontinuous transfers, the driver uses DMA system calls *read()* and *write()*. Each *read()* and *write()* system call allocates kernel resources, during which time DMA transfers are interrupted.

To perform continuous transfers, use the ring buffers. The ring buffers are a set of buffers that applications can access continuously, reading and writing as required. When the last buffer in the set has been accessed, the application then cycles back to the first buffer. See `edt_configure_ring_buffers()` for a complete description of the ring buffer parameters that you can configure. See the sample programs `simple_getdata.c` and `simple_putdata.c` distributed with the driver for examples of using the ring buffers.

Elements of SCD Applications

Applications for performing continuous transfers typically include the following elements:

```
#include "libedt.h"

main()
{
    EdtDev *edt_p = edt_open("scd", 0) ;
    char *buf_ptr; int outfd = open("outfile", 1) ;

    /* Configure a ring buffer with four 1MB buffers */
    edt_configure_ring_buffers(edt_p, 1024*1024, 4, EDT_READ, NULL) ;
    edt_start_buffers(edt_p, 0) ; /* 0 starts unlimited buffer DMA */
    /* This loop will capture data indefinitely, but the write()
     * (or whatever processing on the data) must be able to keep up. */
    while ((buf_ptr = edt_wait_for_buffer(edt_p, 1)) != NULL)
        write(outfd, buf_ptr, 1024*1024) ;

    edt_close(edt_p) ;
}
```

Applications for performing noncontinuous transfers typically include the following elements:

```
#include "libedt.h"

main()
{
    EdtDev *edt_p = edt_open("scd", 1) ;
    char buf[1024] ;
    int numbytes, outfd = open("outfile", 1) ;
    /*
     * Because read()s are noncontinuous, unless is there hardware
     * handshaking there will be gaps in the data between each read().
     */
    while ((numbytes = edt_read(edt_p, buf, 1024)) > 0)
        write(outfd, buf, numbytes) ;

    edt_close(edt_p) ;
}
```

You can use ring buffer mode for real-time data capture using a small number of (typically 1 MB) buffers configured in a round-robin data FIFO. During capture, the application must be able to transfer or process the data before data acquisition wraps around and overwrites the buffer currently being processed.

The example below shows real-time data capture using ring buffers, although it includes no error-checking. In this example, `process_data(bufptr)` must execute in the same amount of time it takes DMA to fill a single buffer, or faster.

```
#include "libedt.h"

main()
{
    EdtDev *edt_p = edt_open("scd", 0) ;

    /* Configure four 1 MB buffers:
     *   one for DMA
     *   one for the second DMA register on most EDT boards
     *   one for "process_data(bufptr)" to work on
     *   one to keep DMA away from "process_data()"
     */
    edt_configure_ring_buffers(edt_p, 0x100000, 4, EDT_READ, NULL) ;
    edt_start_buffers(edt_p, 0) ; /* 0 starts unlimited buffer DMA */
    for (;;)
    {
        char *bufptr ;

        /* Wait for each buffer to complete, then process it.
         * The driver continues DMA concurrently with processing.
         */
        bufptr = edt_wait_for_buffers(edt_p, 1) ;
        process_data(bufptr) ;
    }
}
```

Use the "`-D_REENTRANT -ledt -lthread`" options to compile and link the library file *libedt.a* with your program. See the makefile and example programs provided for examples of compiling programs using the library routines.

DMA Library Routines

The DMA library provides a set of consistent routines across many of the EDT products, with simple yet powerful ring-buffered DMA capabilities. **Table 1, DMA Library Routines** lists the general DMA library routines. In addition, if driver-specific library routines exist, they can be found in a table thereafter.

The sections that follow describe the DMA library routines in an order corresponding roughly to their general usefulness.

Routine	Description
<code>edt_open</code>	Opens the SCD for application access.
<code>edt_close</code>	Terminates access to the SCD and releases resources.
<code>edt_read</code>	Single, application-level buffer read from the SCD.
<code>edt_write</code>	Single, application-level buffer write to the SCD.
<code>edt_configure_ring_buffers</code>	Configures the ring buffers.
<code>edt_disable_ring_buffers</code>	Stops DMA transfer, disables ring buffers and releases resources.
<code>edt_start_buffers</code>	Begins DMA transfer from or to specified number of buffers.
<code>edt_stop_buffers</code>	Stops DMA transfer after the current buffer(s) complete(s).
<code>edt_wait_for_buffers</code>	Blocks until the specified number of buffers have completed.
<code>edt_next_writebuf</code>	Returns a pointer to the next buffer scheduled for output DMA.
<code>edt_check_for_buffers</code>	Checks whether the specified number of buffers have completed without blocking.
<code>edt_wait_for_next_buffer</code>	Waits for the next buffer that completes DMA.
<code>edt_ring_buffer_overrun</code>	Detects ring buffer overrun which may have corrupted data.
<code>edt_buffer_addresses</code>	Returns an array of addresses referencing the ring buffers.
<code>edt_abort_dma</code>	Cancels the current DMA, resets pointers to the current buffer
<code>edt_abort_current_dma</code>	Cancels the current DMA, moves pointers to the next buffer.
<code>edt_done_count</code>	Returns absolute (cumulative) number of completed buffers.
<code>edt_reset_ring_buffers</code>	Stops DMA in progress and resets the ring buffers.
<code>edt_microsleep</code>	Process sleeps for the specified number of microseconds.

Table 1. General DMA Library Routines

edt_open

Description

Opens the specified SCD and sets up the device handle.

Syntax

```
EdtDev *edt_open(char *devname, int unit) ;
```

Arguments

devname a string with the name of the EDT board.

unit specifies the device unit number

Return

A handle of type (`EdtDev *`), or `NULL` if error. (The structure (`EdtDev *`) is defined in `libedt.h`.) If an error occurs, check the `errno` global variable for the error number. The device name for the SCD is "scd". Once opened, the device handle may be used to perform I/O using `edt_read()`, `edt_write()`, `edt_configure_ring_buffers()`, and other input-output library calls.

edt_close

Description

Shuts down all pending I/O operations, closes the device and frees all driver resources.

Syntax

```
int edt_close(EdtDev *edt_p);
```

Arguments

edt_p SCD device handle returned from *edt_open*.

Return

0 on success; -1 on error. If an error occurs, check the *errno* global variable for the error number.

edt_read

Description

Performs a read on the SCD. The UNIX 2 GB file offset bug is avoided during large amounts of input or output, that is, reading past 2^{31} bytes does not fail. This call is not multibuffering, and no transfer is active when it completes.

Syntax

```
int edt_read(EdtDev *edt_p, void *buf, int size);
```

Arguments

edt_p SCD device handle returned from *edt_open*

buf address of buffer to read into

size size of read in bytes

Return

The return value from *read*, normally the number of bytes read; -1 is returned and *errno* is set by *read* on error.

edt_write

Description

Perform a write on the SCD. The UNIX 2 GB file offset bug is avoided during large amounts of input or output; that is, writing past 2^{31} does not fail. This call is not multibuffering, and no transfer is active when it completes.

Syntax

```
int edt_write(EdtDev *edt_p, void *buf, int size);
```

Arguments

edt_p SCD device handle returned from *edt_open*

buf address of buffer to write from

size size of write in bytes

Return

The return value from *write*; -1 is returned and *errno* is set by *write* on error.

edt_configure_ring_buffers

Description

Configures the SBus Configurable DMA Interface ring buffers. Any previous configuration is replaced, and previously allocated buffers are released. Buffers can be allocated and maintained within the SBus Configurable DMA Interface library or within the user application itself.

Syntax

```
int edt_configure_ring_buffers(EdtDev *edt_p, int bufsize, int nbufs,
                             int data_output, void *bufarray[]);
```

Arguments

edt_p SCD device handle returned from *edt_open*

bufsize size of each buffer. For optimal efficiency, allocate a value approximating throughput divided by 20: that is, if transfer occurs at 20 MB per second, allocate 1 MB per buffer. Buffers significantly larger or smaller can overuse memory or lock the system up in processing interrupts at this speed.

nbufs number of buffers. Must be 1 or greater. Four is recommended for most applications.

data_direction Indicates whether this connection is to be used for input or output. Only one direction is possible per device or subdevice at any given time:
EDT_READ = 0
EDT_WRITE = 1

bufarray If NULL, the library will allocate a set of page-aligned ring buffers. If not NULL, this argument is an array of pointers to application-allocated buffers; these buffers must match the size and number of buffers specified in this call and will be used as the ring buffers.

Return

0 on success; -1 on error. If all buffers cannot be allocated, none are allocated and an error is returned. The global variable *errno* is set on error.

edt_disable_ring_buffers

Description

Disables the SBus Configurable DMA Interface ring buffers. Pending DMA is cancelled and all buffers are released.

Syntax

```
int edt_disable_ring_buffers(EdtDev *edt_p);
```

Arguments

edt_p SCD device handle returned from *edt_open*

0 on success; -1 on error. If an error occurs, check the *errno* global variable for the error number.

edt_start_buffers

Description

Starts DMA to the specified number of buffers.

Syntax

```
int edt_start_buffers(EdtDev *edt_p, int bufnum);
```

Arguments

edt_p SCD device handle returned from *edt_open*

bufnum Number of buffers to release to the driver for transfer. An argument of 0 puts the driver in free running mode, and transfers run continuously until **edt_stop_buffers()** is called.

Return

0 on success; -1 on error. If an error occurs, check the *errno* global variable for the error number.

edt_stop_buffers

Description

Stops DMA transfer after the current buffer has completed. Ring buffer mode remains active, and transfers will be continued by calling **edt_start_buffers()**.

Syntax

```
int edt_stop_buffers(EdtDev *edt_p);
```

Arguments

edt_p SCD device handle returned from *edt_open*

Return

0 on success; -1 on error. If an error occurs, check the *errno* global variable for the error number.

edt_wait_for_buffers

Description

Blocks until the specified number of buffers have completed.

Syntax

```
void *edt_wait_buffers(EdtDev *edt_p, int bufnum);
```

Arguments

edt_p SCD device handle returned from *edt_open*

bufnum buffer number for which to block. Completed buffers are numbered cumulatively starting with 0 when the SCD is opened.

Return

Address of last completed buffer on success; NULL on error. If an error occurs, check the *errno* global variable for more information.

NOTE: If the ring buffer is in free-running mode and the application cannot process data as fast as it is acquired, DMA will wrap around and overwrite the referenced buffer. The application must ensure that the data in the buffer is processed or copied out in time to prevent overrun.

For an example of real-time data capture using ring buffers, see the example on page 6.

edt_next_writebuf

Description

Returns a pointer to the next buffer scheduled for output DMA, in order to fill the buffer with data.

Syntax

```
void *edt_next_writebuf(EdtDev *edt_p) ;
```

Arguments

edt_p SCD device handle returned from *edt_open*.

Return

Returns a pointer to the buffer, or NULL on failure. Sets *errno* on failure.

edt_check_for_buffers

Description

Checks whether the specified number of buffers have completed without blocking.

Syntax

```
int edt_check_for_buffers(EdtDev *edt_p, count);
```

Arguments

edt_p SCD device handle returned from *edt_open*.
nbufs number of buffers.
count number of buffers. Must be 1 or greater. Four is recommended.

Return

Returns the address of the ring buffer corresponding to count if it has completed DMA, or NULL if *count* buffers are not yet complete.

NOTE: If the ring buffer is in free-running mode and the application cannot process data as fast as it is acquired, DMA will wrap around and overwrite the referenced buffer. The application must ensure that the data in the buffer is processed or copied out in time to prevent overrun.

edt_wait_for_next_buffer

Description

Waits for the next buffer that finishes DMA. Depending on how often this routine is called, buffers that have already completed DMA might be skipped.

Syntax

```
void *edt_wait_for_next_buffer(EdtDev *edt_p) ;
```

Arguments

edt_p SCD device handle returned from *edt_open*.

Return

Returns a pointer to the buffer, or NULL on failure. Sets *errno* on failure.

edt_ring_buffer_overflow

Description

Returns true (1) when DMA has wrapped around the ring buffer and overwritten the buffer which the application is about to access. Returns false (0) otherwise.

Syntax

```
int edt_ring_buffer_overflow(EdtDev *edt_p);
```

Arguments

edt_p SCD device handle returned from *edt_open*.

Return

1 (true) when overflow has occurred, corrupting the current buffer, 0 false() otherwise..

edt_buffer_addresses

Description

Returns an array containing the addresses of the ring buffers.

Syntax

```
void **edt_buffer_addresses(EdtDev *edt_p);
```

Arguments

edt_p SCD device handle returned from *edt_open*.

Return

An array of pointers to the ring buffers allocated by the driver or the library. The array is indexed from zero to *n*-1 where *n* is the number of ring buffers set in **edt_configure_ring_buffers()**.

edt_abort_dma

Description

Stops any transfers currently in progress, resets the ring buffer pointers to restart on the current buffer.

Syntax

```
int edt_abort_dma(EdtDev *edt_p);
```

Arguments

edt_p SCD device handle returned from *edt_open*.

Return

0 on success; -1 on error. If an error occurs, check the *errno* global variable for the error number.

edt_abort_current_dma

Description

Stops the current transfers, resets the ring buffer pointers to the next buffer.

Syntax

```
int edt_abort_dma(EdtDev *edt_p);
```

Arguments

edt_p SCD device handle returned from *edt_open*.

Return

0 on success; -1 on error. If an error occurs, check the *errno* global variable for the error number.

edt_done_count

Description

Returns the cumulative count of completed buffer transfers in ring buffer mode.

Syntax

```
int edt_done_count(EdtDev *edt_p);
```

Arguments

edt_p SCD device handle returned from *edt_open*.

Return

The number of completed buffer transfers. Completed buffers are numbered consecutively starting with 0 when **edt_configure_ring_buffers()** is invoked. The index of the ring buffer most recently completed by the driver equals the number returned modulo the number of ring buffers. -1 is returned if ring buffer mode is not configured and the global variable *errno* is set.

edt_reset_ring_buffers

Description

Stops any DMA currently in progress, then resets the ring buffer to start the next DMA at *bufnum*.

Syntax

```
int edt_reset_ring_buffers(EdtDev *edt_p, int bufnum) ;
```

Arguments

edt_p SCD device handle returned from *edt_open*.

bufnum The index of the ring buffer at which to start the next DMA.

Return

0 on success; -1 on error. If an error occurs, check the *errno* global variable for the error number.

edt_microsleep**Description**

Causes the process to sleep for the specified number of microseconds.

Syntax

```
int edt_microsleep(u_int usecs) ;
```

Arguments

usecs The number of microseconds for the process to sleep.

Return

0 on success; -1 on error. If an error occurs, check the *errno* global variable for the error number.

Error Conditions

The table below shows error codes for the SCD and the error condition represented by each code.

Error Code	Failing Command	Error Condition
EINVAL	<i>ioctl()</i>	An invalid command was used, or an invalid length was specified for the buffer
EFAULT		An argument points outside the allocated address space.

Table 2. Error Codes and Conditions

Hardware Interface Protocol

This section describes how to connect your device to an SCD interface, including the electrical characteristics of the signal, the signal descriptions, the timing specifications, and the connector pin-out.

Electrical Interface

The SCD uses differential data transmission to transmit data at very high rates over long distances through noisy environments. Differential transmission nullifies the effects of ground shifts and noise. These effects appear as common mode voltages on the transmission line and are rejected by the receiver. A typical balanced differential circuit is shown below.

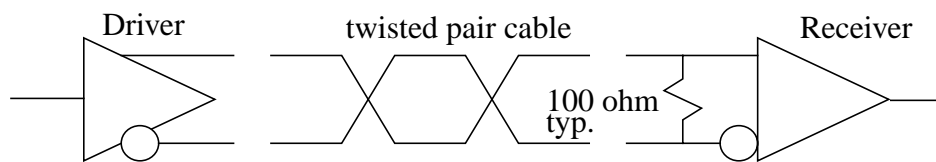


Figure 1. Balanced Differential Circuit

The interface is implemented with 32 signals, each implemented as a differential pair of wires.

RS-422

The SCD-20 DMA interface protocol uses RS-422 signal levels. RS-422 is defined by the Electronic Industries Association to provide robust high-speed data transmission. It allows signaling rates up to 10 MHz for short cables of up to 40 feet and cables of up to 4000 feet at 100 KHz, as shown in the graph below.

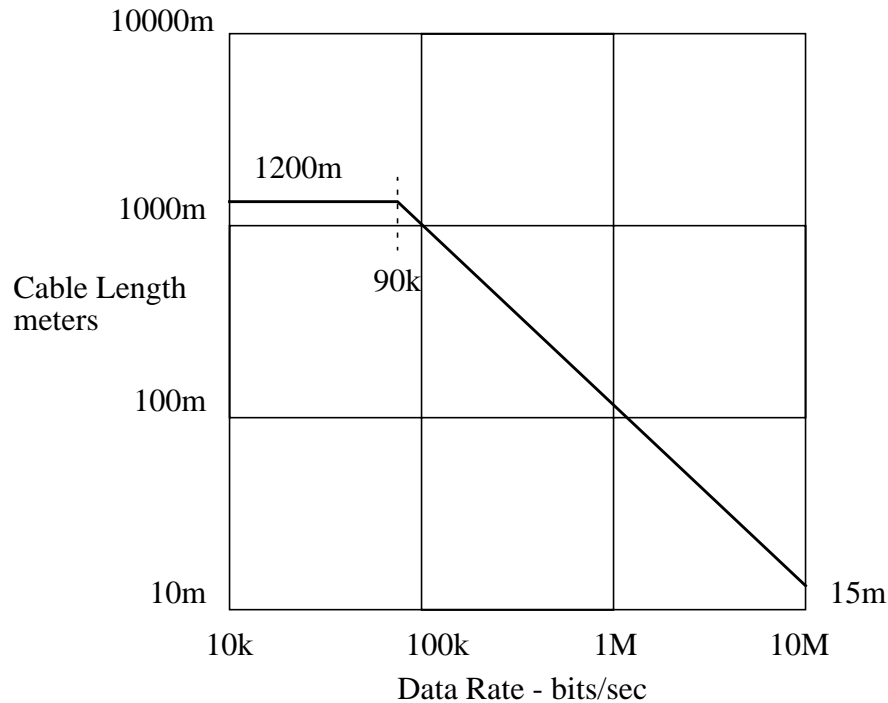


Figure 2. RS-422 Data Signalling Rate and Cable Length

For further information, see the EIA RS-422-A standard, *Electrical Characteristics of Balanced Voltage Digital Interface Circuits*, Dec. 1978, available from the Electronic Industries Association, Engineering Department, 2001 Eye St. N.W., Washington, D. C. 20006.

Pseudo-ECL

The SCD-40 DMA interface protocol uses AT&T pseudo-emitter-coupled logic (PECL) signal levels. Pseudo-ECL levels are ECL levels that are shifted by 5 volts to run on a single +5 volts power supply. These devices are excellent for minimizing electromagnetic interference where data must be transmitted at very high speeds.

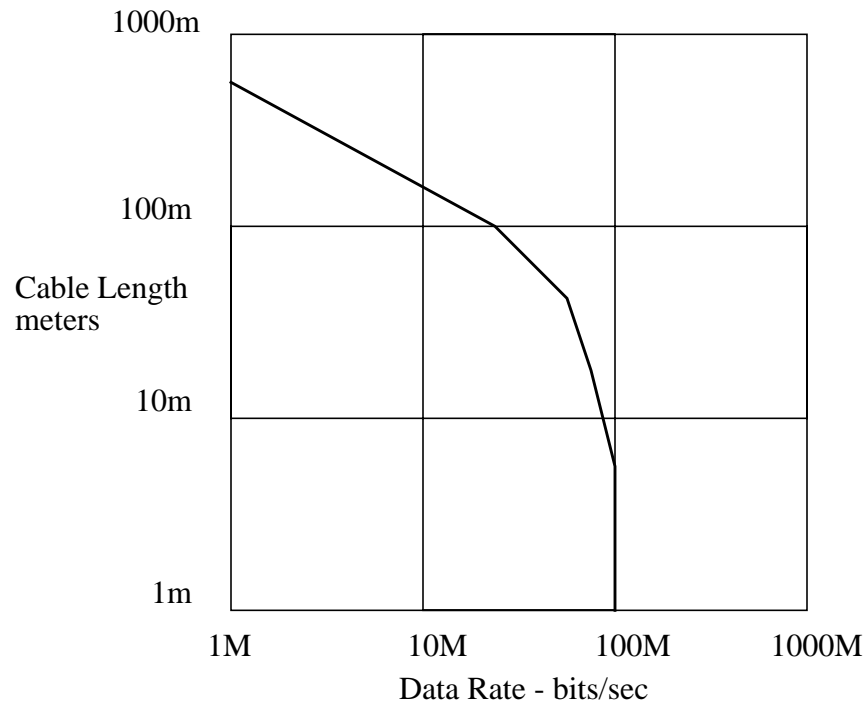


Figure 3. PECL Data Signalling Rate and Cable Length

For further information, see *The 41 Series of High Performance Line Drivers, Receivers, and Transceivers*, Jan. 1991 Data Book and Designers Guide, available from AT&T Microelectronics, CA91-001DBIP.

LVDS

The SCD-60 DMA interface protocol uses low-voltage differential signaling, a new standard for faster signaling at lower power, compatible with IEEE 1596.3 SCI LVDS standard, and conforming to ANSI/TIA/EIA-644-1995 LVDS standard. For information on the relationship between signaling rate and cable length, see TIA/EIA-644 standard, available from Global Engineering Documents, 1-800-854-7179.

Signals

The hardware flow control protocol assumes that FIFO or memory buffers on both ends implement almost-full and almost-empty thresholds. Therefore, when a "not ready to accept data" signal is sent to the transmitting device, the receiver can still accommodate enough data to allow for cable delay and synchronization.

Signal	SCD I/O	Description
DAT(15:0)	I/O	Sixteen bidirectional data lines for DMA data.
STAT(3:0)	I	Four general-purpose control inputs. Any can be enabled to interrupt the SBus host.
FUNCT(3:0)	O	Four general-purpose program control outputs. Can be used to reset the user device or indicate DMA direction for bidirectional devices.
SENDT	O	Send Timing is a constant clock driven by the DMA interface that can be used by the user device to generate the receive timing. This signal does not have to be used. The SCD-20 outputs a 10 MHz clock. The SCD-40 outputs a 20 MHz clock.
RXT	I	Receive Timing is an input to the DMA interface. This is the clock used to synchronize input data and control signals. This signal can be equal to or less than the SENDT frequency of the board in use. It is best, although not required, that this signal is a continuous clock. Data clocked into the DMA interface must propagate through pipelining registers before it can be transferred into SBus memory. If the RXT clock stops, data is left in this pipe instead of being transferred to host memory.
TXT	O	Transmit Timing is an output from the DMA interface. TXT synchronizes the DMA output data and control signals. TXT is either internally generated from the same source as SENDT, or looped back from RXT.
IDV	I	Input Data Valid is asserted by the device synchronous with RXT, to tell the DMA interface that data on the DATA(15:0) signals are valid and must be registered and transferred to the SBus memory. The DMA interface will accomplish this unless the BNR signal has been asserted for >32 IDV signals.
BNR	O	Bus Not Ready is asserted by the DMA interface synchronous with TXT when 32 bytes or fewer of data space remains for input data from the device. This warns you to stop the data transfer or prepare for overflow.
OUTPUT DISABLE	I	Output Disable disables the data outputs when more than one SCD board is connected to the same cable. In order to use this signal, you must set the Enable Output Control bit in the Stat Polarity register. This signal is TTL-compatible.
ODV	O	Output Data Valid is asserted by the DMA interface when it has placed valid output data on DATA(15:0). ODV is asserted synchronously with the TXT clock, and only if the DNR signal is not asserted.
DNR	I	Device Not Ready is asserted by the device synchronous with the RXT clock when the user device is about to run out of space for storing data from the DMA interface. The amount of overrun buffer required in the device varies according to the cable length. The SCD may produce four or more words of valid data after DNR is presented to the input pins.

Table 3. Signals

Timing

The clock and data output timing is specified at the pins of the SCD connector.

	SCD-20	SCD-40	SCD-60
Clock frequency	0–10 MHz	0–20 MHz	0–30 MHz
Clock duty cycle	50% + or –10 ns	50% + or –	50% + or –
Input minimum set-up time	20 ns	5 ns	5 ns
Input minimum hold time	25 ns	6 ns	6 ns
Output maximum propagation delay	20 ns	10 ns	10 ns

Table 4. Timing Specifications

Figure 4 shows the SCD timing:

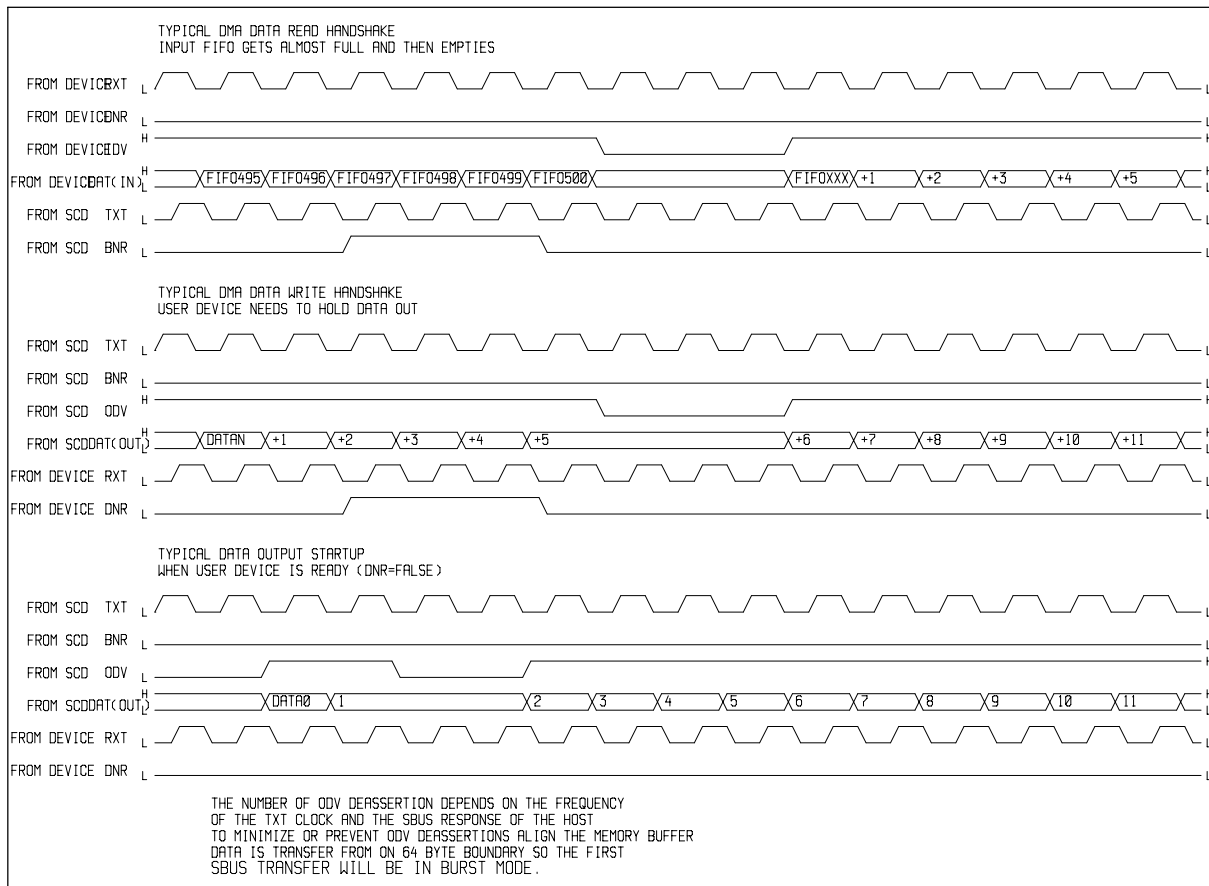


Figure 4. SCD Timing

Connector Pinout

The SCD board uses a high-density 80-pin I/O connector, AMP part number 749111-7, with a straight-shielded backshell (AMP P/N 749196-1) or right angle backshell (AMP P/N 749205-1).

The following pinout diagram describes the connection from the SCD board to the cable.

NOTE: Do not connect your own circuits to the unused pins, as they may be internally connected to the SCD.

AMP	Signal	AMP	Signal
1	Ground	41	Ground
2	Ground	42	Ground
3	DAT4+	43	DAT0+
4	DAT4-	44	DAT0-
5	DAT5+	45	DAT1+
6	DAT5-	46	DAT1-
7	DAT6+	47	DAT2+
8	DAT6-	48	DAT2-
9	DAT7+	49	DAT3+
10	DAT7-	50	DAT3-
11	DAT12+	51	DAT8+
12	DAT12-	52	DAT8-
13	DAT13+	53	DAT9+
14	DAT13-	54	DAT9-
15	DAT14+	55	DAT10+
16	DAT14-	56	DAT10-
17	DAT15+	57	DAT11+
18	DAT15-	58	DAT11-
19	not used	59	not used
20	+5V	60	+5V
21	not used	61	not used
22	OUTPUT DISABLE	62	not used
23	Ground	63	Ground
24	STAT0+	64	RXT+
25	STAT0-	65	RXT-
26	STAT1+	66	IDV+
27	STAT1-	67	IDV-
28	STAT2+	68	DNR+
29	STAT2-	69	DNR-
30	STAT3+	70	reserved+
31	STAT3-	71	reserved-
32	FUNCT0+	72	SENDT+
33	FUNCT0-	73	SENDT-
34	FUNCT1+	74	ODV+
35	FUNCT1-	75	ODV-
36	FUNCT2+	76	BNR+
37	FUNCT2-	77	BNR-
38	FUNCT3+	78	TXT+
39	FUNCT3-	79	TXT-
40	Ground	80	Ground

Table 5. Connector Pinout

Registers

The SCD board is configured and controlled with nine 32-bit registers, two 16-bit registers, two 8-bit registers, and an FCode PROM. Applications can access the SCD registers through the DMA library routines, or if necessary by means of *ioctl()* calls with SCD-specific parameters, as defined in the file *scd.h*.

NOTE: All registers except the Xilinx interface control registers are initialized and manipulated by the SCD driver. User applications do not ordinarily need to read or write these registers.

SBus Addresses

The addresses listed in the table below are offsets from the SBus slot base addresses. Obtain the SBus base address from the SBus host documentation. The figure below describes the SCD board registers in detail.

0x0001.00C0	Direction control		not used	
0x0001.0084	Status polarity			
0x0001.0080	Command	Data path status	Funct	Stat
	not used			
0x0001.0050	Xilinx programming			
0x0001.004C	Control and next count			
0x0001.0048	Current count			
0x0001.0044	Next DMA address			
0x0001.0040	Current DMA address			
	not used			
0x0001.000C	Input FIFO almost full threshold			
0x0001.0008	Input FIFO almost empty threshold			
0x0001.0004	Output FIFO almost full threshold			
0x0001.0000	Output FIFO almost empty threshold			
			ROM byte 0x7FFF	
	ROM			
0x0000.0000	ROM byte 0			
Byte	0	1	2	3
Word	0		1	

Figure 5. SBus Addresses

Output FIFO Almost Empty Threshold Register

The output FIFO almost empty threshold register is a 32-bit register at address 0x10000. This register allows you to specify an 8-bit number representing the number of data items in the output FIFO required to set the "almost empty" flag. The SCD board includes two output FIFOs, and both must be set to the same

threshold with the same write operation. Therefore, bits 0–7 and bits 16–23 must hold the same value. Bits 8–15 and bits 24–31 are not used; set them to zero.

Bit	Description
OE31–24	Set to 0.
OE23–16	The number of data items in the output FIFO required to set the “almost empty” flag.
OE15–8	Set to 0.
OE7–0	The number of data items in the output FIFO required to set the “almost empty” flag.

Table 6. The Output FIFO Almost Empty Threshold

Output FIFO Almost Full Threshold Register

The output FIFO almost full threshold register is a 32-bit register at address 0x10004. This register allows you to specify an 8-bit number representing the number of data items in the output FIFO required to set the “almost full” flag. The SCD board includes two output FIFOs, and both must be set to the same threshold with the same write operation. Therefore, bits 0–7 and bits 16–23 must hold the same value. Bits 8–15 and bits 24–31 are not used; set them to zero.

Bit	Description
OE31–24	Set to 0.
OE23–16	The number of data items in the output FIFO required to set the “almost full” flag.
OE15–8	Set to 0.
OE7–0	The number of data items in the output FIFO required to set the “almost full” flag.

Table 7. The Output FIFO Almost Full Threshold

Input FIFO Almost Empty Threshold Register

The input FIFO almost empty threshold register is a 32-bit register at address 0x10008. This register allows you to specify an 8-bit number representing the number of data items in the input FIFO required to set the “almost empty” flag. The SCD board includes two input FIFOs, and both must be set to the same threshold with the same write operation. Therefore, bits 0–7 and bits 16–23 must hold the same value. Bits 8–15 and bits 24–31 are not used; set them to zero.

Bit	Description
OE31–24	Set to 0.
OE23–16	The number of data items in the input FIFO required to set the “almost empty” flag.
OE15–8	Set to 0.
OE7–0	The number of data items in the input FIFO required to set the “almost empty” flag.

Table 8. The Input FIFO Almost Empty Threshold

Input FIFO Almost Full Threshold Register

The input FIFO almost full threshold register is a 32-bit register at address 0x1000C. This register allows you to specify an 8-bit number representing the number of data items in the input FIFO required to set the “almost full” flag. The SCD board includes two input FIFOs, and both must be set to the same threshold

with the same write operation. Therefore, bits 0–7 and bits 16–23 must hold the same value. Bits 8–15 and bits 24–31 are not used; set them to zero.

Bit	Description
OE31–24	Set to 0.
OE23–16	The number of data items in the input FIFO required to set the “almost full” flag.
OE15–8	Set to 0.
OE7–0	The number of data items in the input FIFO required to set the “almost full” flag.

Table 9. The Input FIFO Almost Full Threshold

Current DMA Address Register

The current DMA address register is a 32-bit read-only register at address 0x10040.

Bit	Description
A31–A22	The 4 MB page addressed by the DMA.
A21–A2	When read, the next address to access on the SBus.
A1–A0	Always 0.

Table 10. The Current DMA Address Register

Next DMA Address Register

The next DMA address register is a 32-bit register at address 0x10044.

Bit	Description
NA31–22	Show or store the 4 MB page addressed by the next DMA. When the next DMA starts, this value is copied into the corresponding bits of the current DMA address register.
NA21–2	Sets the address the next DMA will use. When the next DMA starts, this value is copied into the corresponding bits of the current DMA address register.
NA1–0	Set to 0.

Table 11. The Next DMA Address Register

Current Count Register

The current count register is a 32-bit read-only register at address 0x10048.

Bit	Description
C31–22	Always 0.
C21–C2	When read, these bits display how many words remain in the DMA transfer currently in progress.
C1–0	Always 0.

Table 12. The Current Count Register

Control and Next Count Register

The control and next count register is a 32-bit register at address 0x1004C.

Bit	SCD_	Description
D31	INT	A read-only status bit. A value of 1 indicates the SCD-20 is asserting an SBus interrupt.
D30	DMA_INPROG	A read-only status bit. A value of 1 indicates a DMA in progress.
D29	DMA_START	A value of 1 enables DMA transfer.
D28	EN_INTFC	A value of 1 enables interface interrupt.
D27	EN_EODMA	A value of 1 enables end-of-DMA interrupt.
D26	CANCEL	Cancels SBus DMA in progress.
D25	DMA_DIR_MSK	DMA direction: a value of 1 reads host memory, 0 writes it.
D24	BURST_EN	A value of 1 enables burst transfer.
D23–22		The burst size for burst transfers. Values are mapped as follows:
	2WD_BURST	0 = 2-word burst transfer
	4WD_BURST	1 = 4-word burst transfer
	8WD_BURST	2 = 8-word burst transfer
	16WD_BURST	3 = 16-word burst transfer
D21–2	SIZ_MSK	How many words to transfer in the next DMA transfer. When the next DMA starts, this value is copied into the corresponding bits of the current count register.
D1–0	CNT_MSK	Always 0.

Table 13. The Control and Next Count Register

Xilinx Programming Register

The Xilinx programming register is a 32-bit register at address 0x10050. The Xilinx chip is a programmable integrated circuit used to implement the SCD interface or to test the board. The Xilinx programmable IC is programmed serially.

NOTE: Any registers defined to control the interface reside in the Xilinx IC. In order to access those registers, the SCD requires that the Xilinx be loaded with a program that defines them. If the Xilinx is not loaded, or loaded with an incorrect program, those registers are inaccessible.

The Xilinx IC is programmed when the SCD driver is loaded, or by the application program.

Bit	X_	Description
D31–2		not used
D1	PROG (<i>write</i>) INIT (<i>read</i>)	Sets the Xilinx IC into programming mode, in which it is able to accept program data. To do so: <ol style="list-style-type: none"> 1. Set this bit to 1. 2. Clear this bit to 0. 3. Wait until reading this bit produces a value of 1, indicating that the IC is ready to accept a program.
D0	DATA_MSK (<i>write</i>) DONE (<i>read</i>)	The bit used for the serial data stream containing the program. When read, a value of 1 indicates that the IC is done accepting the program data.

Table 14. Xilinx Programming Register

Direction Control Register

The direction control register is a 16-bit register at address 0x100C0. This register determines whether the physical drivers or receivers on the interface are inputs or outputs. Each pin on the SCD can be programmed as either an input or an output. Those pins described only as inputs become outputs only during tests, and those described only as outputs become inputs only during tests. The SCD driver modifies the data bits appropriately for a *read* or *write* system call. Setting the same pins to serve as both input and output is useful for testing. Setting the same pins to serve as neither input nor output is not useful.

For example, to implement the interface documented in the connector pinout shown on page 21, set this register as follows:

Low bits	4–7, 9, 11, 12, 14
High bits	0–3, 8, 10, 13, 15

To do so, send the value 0x5A0F.

Bit	Description
DC15	Pins 32–39 are inputs when this bit is low.
DC14	Pins 24–31 are inputs when this bit is low.
DC13	Pins 72–79 are inputs when this bit is low.
DC12	Pins 64–71 are inputs when this bit is low.
DC11	Pins 72–79 are outputs when this bit is low.
DC10	Pins 64–71 are outputs when this bit is low.
DC9	Pins 32–39 are outputs when this bit is low.
DC8	Pins 24–31 are outputs when this bit is low.
DC7	Pins 11–18 are inputs when this bit is low.
DC6	Pins 3–10 are inputs when this bit is low.
DC5	Pins 51–57 are inputs when this bit is low.
DC4	Pins 43–50 are inputs when this bit is low.
DC3	Pins 51–57 are outputs when this bit is low.
DC2	Pins 43–50 are outputs when this bit is low.
DC1	Pins 11–18 are outputs when this bit is low.
DC0	Pins 3–10 are outputs when this bit is low.

Table 15. The Direction Control Register

Command Register

The command register is an 8-bit read-write register at address 0x10080.

Bit	SCD_	Description
D0	DIR	A value of 1 indicates that data is coming in to the SCD. A value of 0 indicates that data is going out from the SCD.
D1	not used	
D2	DATA_INV	If this bit is set, the SCD inverts the data.
D3	ENABLE	Set to one to enable the SCD interface. This bit is set after the direction is chosen and typically after the first DMA buffer is ready. To reset direction or flags this bit must be reset. To flush the DMA FIFOs, clear then set this bit.
D4–7	STAT_INT_EN	A value of one enables the corresponding STAT bit to cause an interrupt when it is asserted.

Table 16. The Command Register

Data Path Status Register

The data path status register is an 8-bit read-only register at address 0x10081.

Bit	SCD_	Description
D0	OF_NOT_EMP	If this bit is set, the output FIFO is not empty.
D1	IF_NOT_EMP	If this bit is set, the input FIFO is not empty.
D2	UNDERFLOW	If the DNR signal is low and the ODV signal goes low because the output FIFO runs out of data, then this bit is asserted and remains so throughout the data transfer. Reset this bit with the ENABLE bit in the command register.
D3	OVERFLOW	This bit is asserted when the input FIFO is full and the IDV signal is high. Reset this bit with the ENABLE bit in the command register.
D4–7		ignore - test signals

Table 17. The Data Path Status Register

Stat Register

The stat register is an 8-bit read-only register at address 0x10083.

Bit	SCD_	Description
D0–3	STAT	The state of user-definable STAT input signals as last sampled by the RXT clock signal.
D4–7	STAT_INT	<p>Interrupt bits for the status bits. If the following conditions are both true, then the corresponding bit of these four can be asserted to cause an SBus interrupt:</p> <ul style="list-style-type: none"> The device interrupt is enabled using the EN_INTFC bit in the control and next count register. The corresponding bit is asserted in the command register (one of bits 4–7, named STAT_INT_EN). <p>The SBus interrupt is then caused when the corresponding STAT signal is asserted according to the polarity specified in the stat polarity register. To reset the interrupt, disable and re-enable the appropriate STAT_INT_EN bit in the command register.</p>

Table 18. The Stat Register

Funct Register

The funct register is an 8-bit read-write register at address 0x10082.

Bit	SCD_	Description
D0–3	FUNCT	Sets the state of the user-definable FUNCT outputs.
D4–7		not used

Table 19. The Funct Register

Stat Polarity Register

The stat polarity register is an 8-bit read-write register at address 0x10084.

Bit	SCD_	Description
D0-3	POLARITY	A value of 0 indicates that a change from 0 to 1 from one clock cycle to the next causes an interrupt in the corresponding bit of the STAT_INT register, if the corresponding bit is also enabled in STAT_INT_EN. A value of 1 causes the same event when the STAT_INT bit changes from 1 to 0 from one clock cycle to the next.
D4	STAT_INT_ENA	Provides global enable or disable for all interrupt bits in Stat register, allowing the driver to disable and re-enable them in one operation, without altering the state of the Stat register. This bit is used mainly by the driver to disable the Stat interrupts to determine which other interrupts are pending. A value of 1 enables the interrupts.
D5	ENA_OUT_CTRL	When set, enables the OUTPUT DISABLE signal on pin 22.
D6-7		not used

Table 20. The Stat Polarity Register

Specifications

The SCD Board conforms to the following specifications.

SBus Compliance

Number of slots	1
Transfer size	1, 2, 4, 8, or 16 words per transfer— the largest size accommodated by the host computer
DVMA master	
SBus memory space	approximately 66 KB
Clock rate	25 MHz

Device Data Transfer

Protocol	Synchronous stream
Buffers	SCD-20: 1 KB FIFO for input, 1 KB FIFO for output SCD-40: 2 KB FIFO for input, 2 KB FIFO for output SCD-60: 2 KB FIFO for input, 2 KB FIFO for output

Software

Drivers for Sun OS Version 4.1.x and System V Version 4 (Solaris 2.x)

Power

5 V at 2 A

Environmental

Temperature	Operating: 10 to 40 C C
	Operating: 20 to 80% noncondensing at 40 C C

Physical

Dimensions	3.3" x 5.78" x 0.5"
Weight	3.5 oz.

Appendix A *ioctl()* Parameters

Engineering Design Team recommends that applications use the software library interface documented starting on page 5. However, if necessary, the following *ioctl* parameters are useful for application programs. Others may be defined, but are used for Engineering Design Team's internal purposes only.

When calling *ioctl()* when using *libedt.a*, pass *edt_p->fd* as the first argument, where *edt_p* is returned from *edt_open()*.

EDTx_RTIMEOUT

Set (**S**) or get (**G**) the timeout lengths for reads from the SCD. The timeout units are hundredths of seconds. Provide the address of an unsigned short as the third argument.

EDTx_WTIMEOUT

Set (**S**) or get (**G**) the timeout lengths for writes to the SCD. The timeout units are hundredths of seconds. Provide the address of an unsigned short as the third argument.

SCDS_STATx_SIG

Set the signal to send to the current process upon receipt of a *STAT* interrupt. Replace *x* with an integer 1–4, specifying one of *STAT* bits 1–4. This *ioctl()* parameter enables interrupts so the SCD can receive the attention signal, even when I/O is not in progress. Set the signal to zero to disable the signal upon interrupt. Provide the address of an unsigned integer as the third argument.

EDTx_DEBUG_LEVEL

Set (**S**) or get (**G**) the debug level. A value of 1 reports driver status when *modinfo* (Solaris) or *modstat* (SunOS 4.1.x) is run. Other values are used for purposes internal to EDT. Provide the address of an unsigned short as the third argument.

EDTx_DEBUG_INTR

Set (**S**) or get (**G**) the debug interrupt level—1 generates full driver tracing and may interfere with normal driver operation. Provide the address of an unsigned short as the third argument.

SCDx_OUT_EMPTY, SCDx_OUT_FULL, SCDx_IN_EMPTY, SCDx_IN_FULL

Access FIFO control registers. This should never be done without specific direction from EDT, these registers control the internal operation of the SCD and are set to the proper values for your computer by the driver. These parameters require the address pointed to (the third parameter to *ioctl()*) to contain unsigned integers (32 bits). Provide the address of an unsigned integer as the third argument.

SCDx_COMMAND

Set or get the value of the command register. Typically used to set the data polarity and the transmit clock source. The DMA direction and enable is accomplished by the read and write system calls. Provide the address of an unsigned character as the third argument.

SCDG_DATA_PATH_STATUS

Get the value of the data path status register. This registers reflects the state of the SCD FIFOs. Provide the address of an unsigned character as the third argument.

SCDG_STAT

Get the value of the stat register. These bits represent the state of input signals from the user device. Provide the address of an unsigned character as the third argument.

SCDx_FUNCNT

Set or get the value of the function bits register. Provide the address of an unsigned character as the third argument.

SCDx_STAT_POLARITY

Set or get the value of the stat bits interrupt polarity register. Provide the address of an unsigned character as the third argument.

SCDx_DIR_WHEN_REGISTER

Set or get the value of the named register at the start of or the end of subsequent *read* or *write* system calls. Substitute **S** or **G** for **x**. Substitute **READ** or **WRITE** for **DIR**. Substitute **START** or **END** for **WHEN**. Substitute the register name for **REGISTER**. This word is valid only until the next *open()* of the SCD. Provide the address of an unsigned character as the third argument.

SCD_CLEAR_FUNCT_OPS

Clears all previous registers manipulated by **SCDx_DIR_WHEN_REGISTER**.

SCDS_DEF_SHORT_XFER

Set short transfer timeout mode. Provide the address of an unsigned short as the third argument.

Set **SCDS_DEF_SHORT_XFER** to a nonzero number to eliminate timeout errors. Upon timeout in this mode, *read* or *write* system calls return the size of the transfer instead of -1, and also put the transfer size into the `aio_result` structure for asynchronous I/O.

Set **SCDS_DEF_SHORT_XFER** to a number greater than one to reset the timeout counter at the start of each transfer.

If **SCDS_DEF_SHORT_XFER** is set to one, and either **EDTS_RTIMEOUT** or **EDTS_WTIMEOUT** are set, timeouts start during read or write operations only when input or output is not already in progress. The timeout is canceled if a transfer completes and no other asynchronous requests are queued.

EDTG_CUR_ADDR

Reads the contents of the current DMA address register. Provide the address of an unsigned integer as the third argument.

EDTG_CUR_COUNT

Reads the contents of the current count register. Provide the address of an unsigned integer as the third argument.

EDTG_NEXT_ADDR

Reads the contents of the next DMA address register. Provide the address of an unsigned integer as the third argument.

EDTG_NEXT_COUNT

Reads the contents of the next count register. Provide the address of an unsigned integer as the third argument.

EDTx_DEBUG_LEVEL Set (**S**) or get (**G**) the debug level. A value of 1 reports driver status when `modinfo` (Solaris) or `modstat` (SunOS 4.1.x) is run. Other values are used for purposes internal to EDT. Provide the address of an unsigned short as the third argument.

EDTx_DEBUG_INTR

Set (**S**) or get (**G**) the debug interrupt level—1 generates full driver tracing and may interfere with normal driver operation. Provide the address of an unsigned short as the third argument.

EDTS_TERMINATE

Shut down ring buffers and terminate DMA on all outstanding I/O requests. No third argument is needed.

EDTS_EODMA_SIG

Register an end-of-DMA signal when the next DMA operation has been completed. The third argument to the *ioctl* is the address of an unsigned integer specifying the number of the signal to send to the calling process. (SIGIO is recommended, as its purpose is to signal an I/O event.) This registration produces one occurrence of a signal; the signal handler must reregister each time a signal is required. Provide the address of an unsigned short as the third argument.

EDTS_NBUFIO Initiates continuous ring-buffer mode. The third argument to the *ioctl* is the address of an unsigned integer specifying the number of buffers in the ring; legal values are between 1 and 40. The driver waits until it has received the specified number of requests for DMA operation, then allocates operating system resources to each buffer, and finally performs continuous DMA to each buffer on a round-robin basis, starting with the first request and wrapping to the beginning again after the last. Provide the address of an unsigned integer as the third argument.

EDTG_NBUFIO Returns the number of the buffer that has most recently completed DMA and is available for processing. Provide the address of an unsigned integer as the third argument.

EDT_FREE_BUF Turns off continuous ring-buffer mode. No third argument is needed.

EDTG_DONECOUNT**EDTG_DEVBUFS_COMPLETED**

Use with continuous ring-buffer mode (**EDTS_NBUFIO**). Get the count of the last buffer completed by the driver. The count starts at 0 and increments each time DMA on a buffer completes. Provide the address of an unsigned integer as the third argument.

EDTS_WAKEUP_DONECOUNT

Use with continuous ring-buffer mode (**S11S_NBUFIO**) and **S11G_DEVBUFS_COMPLETED**. This *ioctl* does not return until the count of buffers completed reaches the count supplied in the third argument to the *ioctl*. This causes the application to wait until the driver has performed the specified number of DMA operations. If the specified count has already been reached it returns immediately. Provide the address of an unsigned integer as the third argument.

The driver counter wraps at 2^{32} , and the driver is implemented to handle this behavior correctly. Therefore let the count in your application wrap as well.

EDTS_LOCKSTEP

Use with continuous ring-buffer mode (**S11S_NBUFIO**). Initiates lockstep mode, in which the driver waits for notification from the application before proceeding with DMA. The number of buffers processed before waiting are specified in the value of the address pointed to by the third argument to this *ioctl*. The driver then waits until it receives notification by means of the following *ioctl*. Provide the address of an unsigned integer as the third argument.

EDT_LOCKOFF Turns off lockstep mode. No third argument is needed.

EDTS_APPBUFS_COMPLETED

Use with continuous ring-buffer mode (**S11S_NBUFIO**) and lockstep mode (**S11S_LOCKSTEP**). Instructs the driver to process the number of buffers as specified in the third argument to this *ioctl*, then wait until this *ioctl* is called again. Provide the address of an unsigned integer as the third argument.

EDTG_BYTECOUNT

Get the value of the *bytecount* variable in the driver—a variable used to count the number of bytes of DMA performed since the last time the value of *bytecount* was set. This parameter can help you keep track of how DMA is progressing. Provide the address of an unsigned integer as the third argument.

NOTE: When the value of *bytecount* reaches 4 GB, it is reset to 0. To be certain that the value is valid, therefore, use **EDTS_BYTECOUNT** to reset it to 0 before each DMA operation.

EDTS_BYTECOUNT

Set the value of the *bytecount* variable in the driver. Provide the address of an unsigned integer as the third argument.

EDTG_WAITRDY

In ring buffer mode, ensures that all DMA resources are allocated before returning. No third argument is needed.