

# S16D

SBus High Speed 16-bit I/O Interface  
for Sun SPARC and UltraSPARC computers

## USER'S GUIDE

008-00133-10



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](mailto:info@edt.com)  
Web: [www.edt.com](http://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.

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

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

# Contents

Overview .....	1
Installation.....	2
Installing the Hardware.....	2
Installing the Software .....	3
Using SunOS Version 4.1.x .....	3
Using Solaris 2.x .....	4
Installing the Sample Programs .....	4
Included Files .....	5
Testing.....	6
Input and Output .....	7
Elements of S16D Applications .....	7
DMA Library Routines .....	8
Error Conditions .....	17
Hardware.....	18
SBus Interface .....	18
FIFO .....	18
Device Interface .....	18
Logic Levels .....	18
FCode Capability .....	19
Signals .....	20
Connector Pinout .....	20
Handshake Signals .....	22
Data Signals .....	22
Synchronous Control Signals.....	23
Asynchronous Control Signals.....	23
Timing .....	24
Using the Signals for Data I/O .....	25
Registers.....	26
SBus Addresses .....	26
Command Register .....	26
Status Register .....	28
Configuration Register.....	29
DMA Address Register .....	30
DMA End Address Register.....	30
Specifications .....	31
SBus Compliance .....	31
Device Data Transfer.....	31
Software .....	31
Power.....	31
Environmental.....	31
Physical .....	31
References.....	32
Contacting EDT .....	33

Appendix A ioctl() Parameters.....34

## Figures

Block Diagram of the S16D Interface.....	18
The Drivers .....	19
Timing Diagram .....	24
SBus Addresses .....	26

## Tables

Loopback Diagnostic Tests .....	6
General DMA Library Routines .....	9
Error Codes and Conditions .....	17
Connector Pinout.....	21
Handshake Signals .....	22
Data Signals .....	22
Synchronous Control Signals.....	23
Asynchronous Control Signals.....	23
Timing Specifications .....	24
The Command Register .....	26
The Status Register .....	28
The DMA Address Register.....	30
The DMA End Address Register .....	30

## Overview

The S16D 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. The S16D features 8 KB of FIFO storage in each direction and can support continuous data rates of up to 10 MB per second. The board also includes diagnostic capability.

The S16D fully supports Direct Virtual Memory Access (DVMA) and the requirements of the SunOS operating system. It includes a SunOS-compatible software driver, enabling applications to access the S16D and transfer data across the S16D interface using standard UNIX system calls.

The S16D provides 16 data inputs and 16 data outputs for direct memory access (DMA). The interface also provides three status inputs, three control outputs and a device interrupt. If you use the S16D DMA input, the 16 data outputs are available for additional control signals. If you use the S16D for DMA output, you can use the 16 data inputs for additional device status inputs.

The S16D uses a simple clock/acknowledge protocol for the DMA data. Handshake polarity is programmable.

A PCI Local bus version of the S16D, with drivers for Solaris, Solaris X/86, or Windows NT is also available. Contact your distributor or EDT for details.

This document describes how to install the S16D interface and write applications for it. It is divided into the following sections:

<b>Installation</b>	describes how to install the S16D module and its related software.
<b>Testing</b>	describes how to test the S16D module.
<b>Input and Output</b>	describes the <i>ioctl</i> parameters, asynchronous input and output, the <i>edtlb</i> dma library, and error conditions.
<b>Hardware</b>	explains the interfaces to the SBus and the device, the FIFOs, logic levels, and FCode for the PROMs.
<b>Registers</b>	describes the S16D hardware registers.
<b>Signals</b>	provides a connector pin-out diagram and describes the S16D signals and timing. It also discusses using the signals for data input and output.
<b>Specifications</b>	lists the specifications.
<b>References</b>	refers to other documents that may prove useful to you when writing applications for the S16D.
<b>Contacting EDT</b>	How to contact us

# Installation

Installing the S16D 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 S16D. Hardware installation is described in the following section. Software installation is described in the section after.

## Installing the Hardware

The S16D is a single-slot SBus board. To install it, refer to your SBus host computer documentation for complete information on installing an SBus board. For example, for the following SPARCstation models this information can be found in these locations:

SPARCstation Model	Location in <i>SPARCstation Installation Guide</i>
1	Appendix A: Installing Boards, Cards, and Modules
1+	Appendix A: Installing SBus Boards and SIMMs
IPX	Appendix C: Installing SBus Cards
10	Appendix B: Opening and Closing the System Unit and <i>Installing SBus Cards in Desktop SPARCstations</i> Sun part no. 800-6635-11.

Use the following procedure to install the S16D:

---



---

### CAUTION

Both the S16D and your SBus host computer contain static-sensitive components. Install the S16D 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 S16D static-shielded bag.
- 
- 

1. Unpack the S16D from the shipping packaging. Do not remove the S16D 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 S16D in the SBus host, following the directions provided with the SBus host. The S16D can be installed in any DMA slot.
3. The S16D is delivered with the jumper installed. Remove this jumper if your device requires the DACK polarity to be positive true from initial host power-up. To do so, lift the black case away from both pins of the jumper, and replace it over just one pin.

To remove the S16D, reverse the installation procedure.

The S16D connects to your device with up to 50 feet of cable. The standard EDT device cable terminates at the S16D with a high-density connector, and at the other end in two standard .100 x .100 40-pin connectors.

EDT provides several cable options for connecting the S16D to an external device. We recommend that you use one of these specially designed cables to achieve the best possible performance. Contact your distributor or EDT for information and specifications for these cables. Each cable is shipped with instructions for connecting the S16D to an external device.

## Installing the Software

The S16D can run on a Sun workstation using either SunOS Version 4.1.x or Solaris 2.x (System V Version 4, or SVR 4). The installation procedures differ. Both are given below.

### Using SunOS Version 4.1.x

If you are using SunOS Version 4.1, use the following procedure to install the S16D driver:

1. Become root or superuser.
2. Change to the directory in which you wish to install the S16D driver.
3. Place the diskette that came with the S16D into the diskette drive.
4. The S16D 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 S16D diskette contains versions of the S16D 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 S16D driver.

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

```
Automatically load the s16d driver during each reboot? [y|n] (y):
```

Entering *y* (or simply typing <Return>) causes the S16D 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 s16d devices do you want? (1):
```

You can install as many S16D boards in your system as you have DMA SBus slots available. Enter the number corresponding to the number of S16D boards you have installed in your system. If you simply type <Return>, one driver is installed.

**NOTE: If you anticipate installing more than one S16D board into your system, install as many S16D drivers as you will ultimately require. The extra drivers will do no harm and will be there when you need them, saving you a step.**

9. If the S16D module 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 S16D driver:

1. Change to the directory in which you placed the S16D files, if necessary.
2. Become root or superuser.
3. Enter:

```
make unload
```

## Using Solaris 2.x

If you are using Solaris 2.x, use the following procedure to install the S16D driver:

1. Become root or superuser.
2. Place the diskette that came with the S16D into the diskette drive.
3. Tell the Solaris volume management to check for a new floppy in the drive, by running

```
volcheck
```

(Note: if you are running File Manager, it may bring up a "Format" pop-up at this point, alerting you to the presence of an unformatted floppy. Dismiss this window.)

4. At the shell prompt, enter:

```
pkgadd -d /vol/dev/aliases/floppy0 EDTs16d
```

To remove the S16D driver:

1. Become root or superuser.
2. Enter:

```
pkgrm EDTs16d
```

For further details, consult your Solaris 2.x documentation, or call the Engineering Design Team.

## Installing the Sample Programs

To install 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

The S16D driver release diskette contains the following files (see the *readme* file for a complete, up-to-date listing):

s16d.o.sun4c	The executable S16D driver for SunOS 4.1.3 on a Sun 4C architecture such as a SPARCStation 1, 1+, 2, or IPC.
s16d.o.sun4m	The executable S16D driver for SunOS 4.1.3 on a Sun 4M architecture such as a SPARCStation 5, 10, 20, LX, Classic, or an Ultra 1 or 2.
s16d	The executable S16D driver for Solaris Version 2.x.
s16d.h	The S16D driver header file, defining <i>ioctls</i> and registers.
drv_ioctl.h	More <i>ioctls</i>
s16d.INSTALL	The installation script used by the makefile (4.1.x only)
makefile	The makefile for installing, loading, and unloading the S16D driver, and making example programs. Used with the SunOS <i>make</i> command to automatically install the driver or compile the example programs.
README	An ASCII file containing last-minute information about the S16D software.
setdebug.c	This file sets the debug levels for the S16D.  Debug level number: <ol style="list-style-type: none"> <li>enables <i>modstat(8)</i> in verbose mode (<i>modinfo</i> for Solaris V. 2.x)</li> <li>provides additional debug information</li> <li>traces the start and end of routines</li> </ol> <p>Interrupt debug level number 1 shows interrupts and DMA setup.</p>
loopback loopback.c	A set of software diagnostics for the S16D board and driver. Use this test with a loopback connector. See the <i>Testing</i> section of this document for more information.
simple_getdata simple_getdata.c	Simple example program illustrating asynchronous I/O read requests to the S16D using ring-buffering and the driver handshaking mechanisms. <i>simple_getdata.c</i> needs an external data source such as a simple data clock.
simple_putdata simple_putdata.c	simple example program illustrating asynchronous I/O write requests from the S16D using ring-buffering and the driver handshaking mechanisms.
s16pio s16pio.c	An example program illustrating programmed I/O using <i>ioctls</i> .
s16piomem s16piomem.c	An example program illustrating programmed I/O performed by mapping registers into application memory.
s16d.c	The source for the S16D driver. This file is present only if you purchased the source from EDT.
libedt.a libedt.c	The DMA library, containing ring-buffered library routines
version	Contains the version number and date of the driver/software package release.

## Testing

The S16D is tested at the factory with a device that reads and writes the interface based on the state of the DIR (direction) line.

You can perform diagnostics on the S16D by installing an optional loopback connector and executing the `loopback.c` diagnostic program included with the standard S16D software. The diagnostic program requires that the S16D driver be installed.

To test the S16D, loop output data from the SPARC back panel or the end of the device cable back to the input. The optional EDT S16D Loopback Kit, part number 012-00067, provides test connectors for both loopback configurations. The loopback kit contains complete instructions for the S16D loopback diagnostics.

The diagnostic program `loopback.c`, used in conjunction with the loopback test connectors, performs the tests described in the table below. To install the program, enter:

```
make loopback
```

Test Name	Test Operation Performed
<b>Programmed I/O Loopback Data Test</b>	Write and read walking ones and zeroes (with each bit uniquely high and low), then read each word and compare the expected and actual values.
<b>Block Loopback</b>	Write and read walking ones and zeroes (with each bit uniquely high and low) using DMA, then read each word and compare the expected and actual values.
<b>Function/Status Bits</b>	Output all function bits, and compare the reported function bits in the Status Register with the looped-back status bits.
<b>DINT signal</b>	Tell the driver to get a <i>SIGIO</i> signal on DINT, then pulse DIR (which loops back to ATTN), and check for occurrence of the <i>SIGIO</i> signal.
<b>Error on DINT</b>	Tell the driver to give an <i>EINTR</i> error if DINT comes in between I/O requests; pulse DIR; then perform a read and check for an error return.
<b>5-second read timeout</b>	Ask the driver to give a 5-second read timeout, then check for the duration and an <i>ETIMEDOUT</i> error return.
<b>5-second write timeout</b>	Ask the driver to give a 5-second write timeout, then check for the duration and an <i>ETIMEDOUT</i> error return.

**Table 1. Loopback Diagnostic Tests**

See the sections entitled *Registers* and *Signals* for further information.

## Input and Output

The S16D 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 S16D Applications

Applications for performing continuous transfers typically include the following elements:

```
#include "libedt.h"

main()
{
    EdtDev *edt_p = edt_open("s16d", 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("s16d", 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("s16d", 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.

<b>Routine</b>	<b>Description</b>
<code>edt_open</code>	Opens the S16D for application access.
<code>edt_close</code>	Terminates access to the S16D and releases resources.
<code>edt_read</code>	Single, application-level buffer read from the S16D.
<code>edt_write</code>	Single, application-level buffer write to the S16D.
<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 2. General DMA Library Routines**

## **edt\_open**

### **Description**

Opens the specified S16D 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 S16D is "s16d". 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*            S16D 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 S16D. 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*            S16D 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 S16D. 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*            S16D 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 High Speed 16-bit I/O Interface ring buffers. Any previous configuration is replaced, and previously allocated buffers are released. Buffers can be allocated and maintained within the SBus High Speed 16-bit I/O 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*            S16D 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 High Speed 16-bit I/O Interface ring buffers. Pending DMA is cancelled and all buffers are released.

### **Syntax**

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

### **Arguments**

*edt\_p*            S16D 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*            S16D 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*            S16D 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*            S16D device handle returned from *edt\_open*

*bufnum*          buffer number for which to block. Completed buffers are numbered cumulatively starting with 0 when the S16D 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 8.

## **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*            S16D 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*            S16D 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*            S16D device handle returned from *edt\_open*.

### **Return**

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

## **edt\_ring\_buffer\_overrun**

### **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_overrun(EdtDev *edt_p);
```

### **Arguments**

*edt\_p*            S16D device handle returned from *edt\_open*.

### **Return**

1 (true) when overrun 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*            S16D 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*            S16D 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*            S16D 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*            S16D 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*            S16D 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 S16D and the error condition represented by each code.

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

**Table 3. Error Codes and Conditions**

## Hardware

This section describes the S16D interface, registers, connectors, and timing. Figure 1 shows a block diagram of the S16D interface.

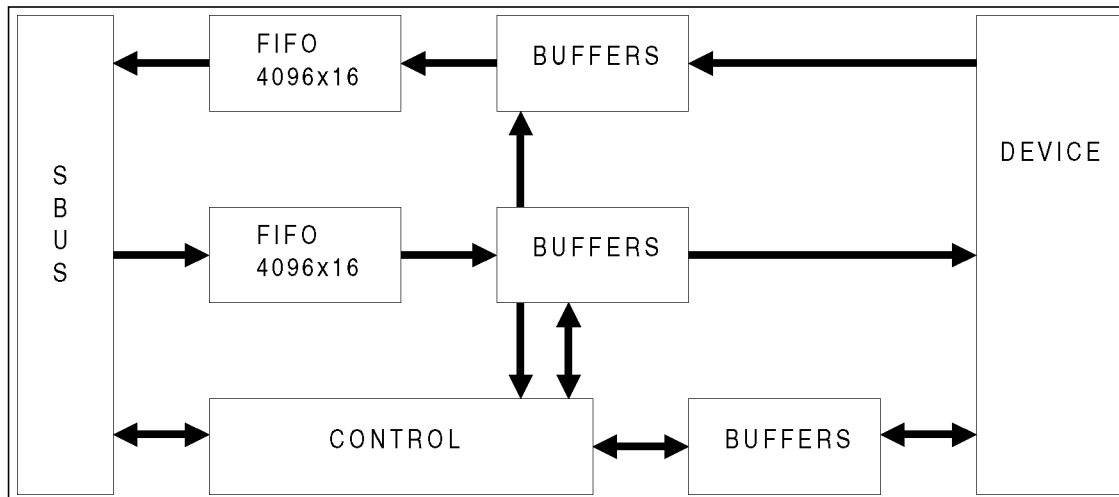


Figure 1. Block Diagram of the S16D Interface

### SBus Interface

The interface to the SBus supports data transfer at 2, 4 and 16 bytes per request. The interface is implemented using programmable logic and high-speed register files. The SBus DMA address is maintained in an application-specific integrated circuit (ASIC).

### FIFO

The S16D uses First-In-First-Out (FIFO) memories to buffer the data flow to and from the SBus. These FIFOs can store up to 8 KB.

### Device Interface

The device interface is implemented with Unibus Driver/Receivers and 180/390 terminators. The receivers have a 1 V hysteresis and a 2 V noise immunity. The drivers are open-collector type. The device handshake and control are implemented in the ASIC.

See the *Signals* section for further details on the device signal usage.

### Logic Levels

The S16D uses the DEC Unibus drivers and receivers. These parts have Schmitt trigger inputs and a switching threshold set to equalize high and low noise margins. You can use TTL devices for short distances, but we do not recommend it. The drivers are inverting open-collector drivers, capable of driving the required 123- $\Omega$  terminating networks at each cable end.

The recommended parts for the drivers are:

- the National Semiconductor DS8838 Quad Driver/Receiver, and
- the National Semiconductor DS8837 Hex Receiver.

Figure 2 shows the configuration of the S16D drivers.

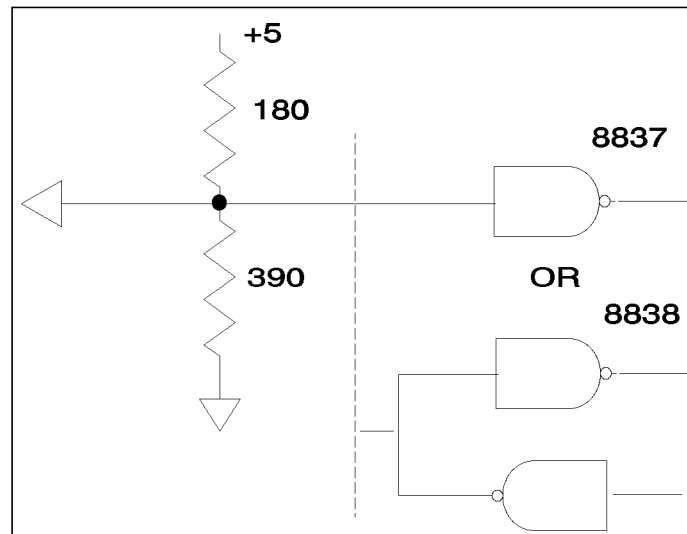


Figure 2. The Drivers

## FCode Capability

The S16D carries a 32 KB x 8-bit PROM mapped into the SBus slot address space at offset zero. By default, this PROM contains identification and initialization parameters, but it can also contain code to initialize the user device. If you require custom FCode programming, contact Sun Microsystems for documentation and EDT for assistance.

## Signals

This section describes the kinds of signals the S16D uses, how they are connected, and provides you with a suggestion for using the signals in your application for data input and output.

### Connector Pinout

The S16D uses a high-density 80-pin I/O connector. The pinout and construction of this connector adapts easily to standard 40-pin, .100 x .100 connectors.

The high-density mating connector is an AMP 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 pinout described in the table below ensures that the high density connector and the P1 and P2 connectors of the EDT cable mate directly with standard 40-pin connectors.

Interpret the connector pinout table below in one of the following ways, depending upon the type of cable you are using.

#### EDT Model CAB-A

The column labeled AMP represents the AMP connector at one end of the cable. This end plugs into the AMP connector on the S16D. The columns labeled STD P1 and STD P2 represent standard 40-pin connectors at the ends of the Y on the other end of the cable. These ends plug into the user device.

#### EDT Model CAB-B

Both ends of the cable have AMP connectors, so the column labeled AMP represents both ends. The columns labeled STD P1 and STD P2 are irrelevant. Cabling is swapped; AMP 1 at one end connects to AMP 41 at the other, AMP 2 to AMP 42, and so on until AMP 40 at one end connects to AMP 80 at the other. This cable can connect two S16D modules, or one S16D to one S11W.

#### EDT Model CAB-D

Both ends of the cable have AMP connectors, so the column labeled AMP represents both ends. The columns labeled STD P1 and STD P2 are irrelevant. Cabling is straight through.

AMP	STD P1	Signal	AMP	STD P2	Signal
1	1	DO15	41	1	DI15
2	2	DO00	42	2	DI00
3	3	DO14	43	3	DI14
4	4	DO01	44	4	DI01
5	5	DO13	45	5	DI13
6	6	DO02	46	6	DI02
7	7	DO12	47	7	DI12
8	8	DO03	48	8	DI03
9	9	DO11	49	9	DI11
10	10	DO04	50	10	DI04
11	11	DO10	51	11	DI10
12	12	DO05	52	12	DI05
13	13	DO09	53	13	DI09
14	14	DO06	54	14	DI06
15	15	DO08	55	15	DI08
16	16	DO07	56	16	DI07
17	17	GROUND	57	17	NC
18	18	GROUND	58	18	GROUND
19	19	RSVD IN	59	19	GROUND
20	20	GROUND	60	20	GROUND
21	21	RDSTRBL	61	21	OUTVALIDL
22	22	GROUND	62	22	GROUND
23	23	STATUS C	63	23	FNCT1
24	24	GROUND	64	24	GROUND
25	25	STATUS C	65	25	RSVD IN
26	26	GROUND	66	26	GROUND
27	27	STATUS B	67	27	FNCT2
28	28	GROUND	68	28	GROUND
29	29	DEVINIT L	69	29	RSVD IN
30	30	GROUND	70	30	GROUND
31	31	STATUS A	71	31	FNCT3
32	32	NC	72	32	FNCT3
33	33	RSVD IN	73	33	RSVD IN
34	34	GROUND	74	34	GROUND
35	35	SDMAEN L	75	35	RSVD IN
36	36	GROUND	76	36	GROUND
37	37	DMAINPUT L	77	37	DEVINT L
38	38	GROUND	78	38	GROUND
39	39	DCLKL P	79	39	DACK P
40	40	GROUND	80	40	GROUND

Table 4. Connector Pinout

The table below describes each signal by name, I/O type, and polarity. An I in the table indicates the signal is an input to the S16D, and an O indicates an S16D output. An H indicates the signal performs the function described in the table at a logic high (or +3 volts). An L indicates the signal performs the named function at a logic low. A P indicates the signal is programmable.

## Handshake Signals

These five signals perform the S16D transfer cycle.

Name	I/O	Assert	Description
DCLK	I	P	The device asserts DCLK to initiate a transfer. The edge of DCLK is used to transfer data, and the DCLK pulse must be at least 80 ns wide. The polarity of the DCLK edge is determined by the CLKP bit in the Configuration Register.
DACK	O	P	DACK is the data transfer acknowledge. DACK is asserted by the S16D in response to a DCLK, and is negated when the transfer is complete. The polarity of DACK assertion is controlled by the DACKP bit in the Configuration Register.
DMAINPUT	O	L	DMAINPUT is asserted low when the S16D is programmed to receive DMA input data on the DI[0-15] inputs.
OUTVALID	O	L	OUTVALID is asserted low when data from the DO[0-15] outputs is valid. In DMA input mode you can use OUTVALID to qualify outputs for programmed I/O. In DMA output mode, OUTVALID indicates you can clock out valid data with DCLK.
SDMAEN	O	L	SDMAEN is asserted low during the actual DMA transfers on the SBus. You can use SDMAEN to qualify DMAINPUT in applications in which the transfer direction changes.
RSTRB	O	L	RSTRB is an 80 ns (min) pulse which indicates the SBus host has read the DIN signals under program control..

**Table 5. Handshake Signals**

## Data Signals

The table below describes the 32 signals that transfer the data. Inputs and outputs are separate for maximum flexibility. The drivers are open-collector, so you can tie the input to the output for a bidirectional bus. For bidirectional operation, write a value of 1 to any output you want to use as a bus input. To use the entire output register as an input bus, write the value FFFF (hexadecimal) to the output register to turn off all output drivers.

Name	I/O	Assert	Description
DIN[0-15]	I	H	Input data
DOUT[0-15]	O	H	Output data

**Table 6. Data Signals**

## Synchronous Control Signals

The table below describes the six signals controlling the synchronous DMA transfer cycle. Each signal controls a device or indicates device status. Devices can implement these signals as required for device applications.

Name	I/O	Assert	Description
FNCT1	O	H	Output for device control, set with FNCT1 in command register
FNCT2	O	H	Output for device control, set with FNCT2 in command register
FNCT3	O	H	Output for device control, set with FNCT3 in command register
STATA	I	H	Input for device status, read with STATA in status register
STATB	I	H	Input for device status, read with STATB in status register
STATC	I	H	Input for device status, read with STATC in status register

**Table 7. Synchronous Control Signals**

## Asynchronous Control Signals

The table below describes the two signals controlling the device and host operating state, asynchronous with the control signals.

Name	I/O	Assert	Description
DEVINT	I	L	Interrupt the SBus host if this signal is enabled in the configuration and command registers. Sensitive to negative edge.
DEVINIT	O	L	If used, DEVINIT is typically connected in the user device and used to reset the device from the S16D command register.

**Table 8. Asynchronous Control Signals**

## Timing

Figure 3 shows the timing diagram for the S16D interface, as specified in the table following. The timing parameters shown in the diagram and table refer to the S16D during DMA transfers, in response to *read* or *write* system calls.

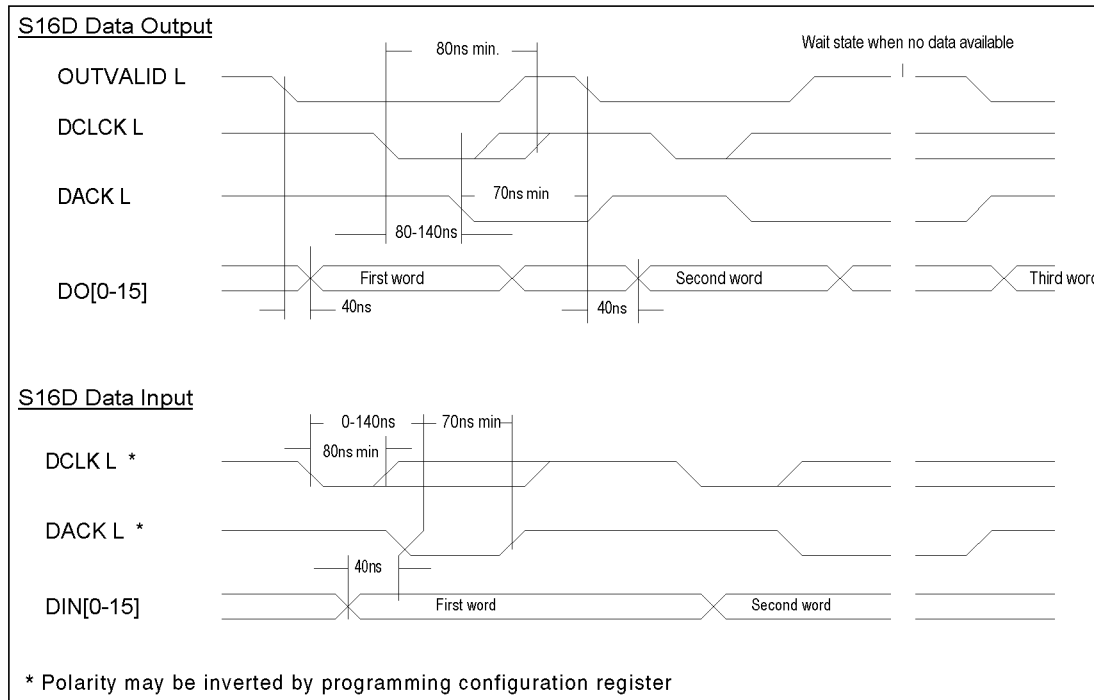


Figure 3. Timing Diagram

Parameter	Min	Max
DCLK Pulse Width	80 ns	DC
DACK Pulse Width	70 ns	
DCLK Assert to DACK Assert	80 ns	140 ns
OUTVALID Pulse Width	30 ns	
DOUT Valid after DACK false	0	40 ns
DIN Setup to DCLK assertion		-40 ns

Table 9. Timing Specifications

## Using the Signals for Data I/O

You can implement an interface handshake in a programmed logic device. Use the following steps to set up the interface and the properties of the resulting handshake:

1. Pulse DEVINIT low to initialize the interface.
2. Determine the DMA direction from the state of DMAINPUT at the first high-to-low SDMAEN transition.
3. To input data to the S16D, assert the first DCLK as soon as the data is available from the device. The data has a negative setup time with DCLK, so you can output data with DCLK while the device generates the next data.
4. The first output data from the S16D, if any, is valid on the DOUT signals 40 ns after OUTVALID goes low. You can assert DCLK as soon as the device accepts this data.
5. If the input FIFO becomes full, or the output FIFO becomes empty, the DACK signal stays low until the FIFO or data are again available. The OUTVALID signal pulses high approximately 30 ns between data words as long as new data is available. Over long cables, the OUTVALID pulse can be too short to use as a handshake signal; use DACK to qualify data over long cables.
6. Use DINT from the device to direct the S16D to terminate DMA, or use the DINIT or FUNC signals from the S16D to direct the device to terminate transfers.

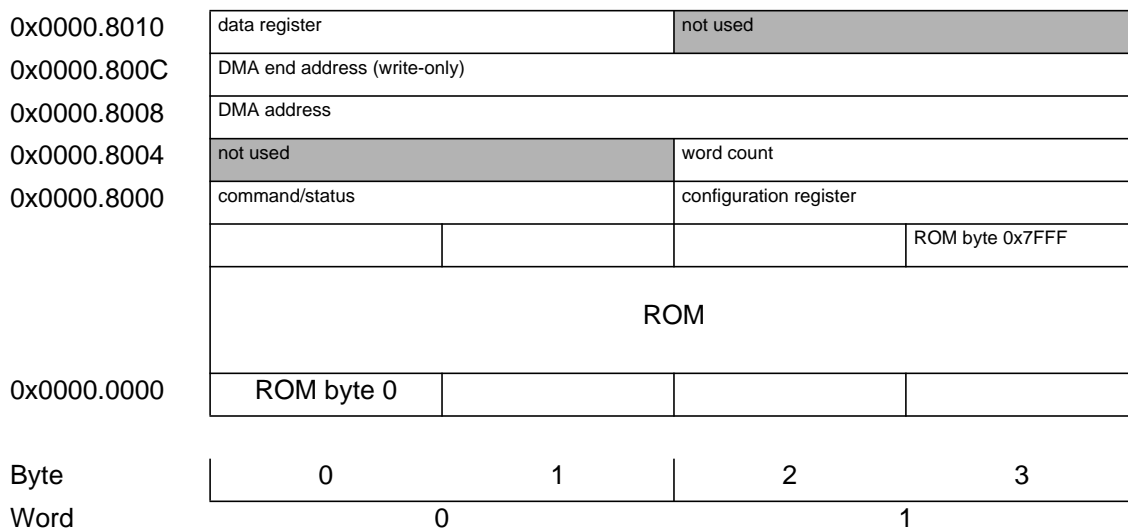
## Registers

The S16D is configured and controlled with two 16-bit registers and two 32-bit registers, plus a data register and an FCode PROM.

Applications can access the S16D registers through the DMA library routines, or if necessary by means of *ioctl()* calls with S16D-specific parameters, as defined in the file *s16d.h*.

## 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 S16D interface registers in detail.



**Figure 4. SBus Addresses**

## Command Register

The command register is a 16-bit write-only register at address 0x8000.

Bit	S16D_	Description
D15	DISINT	Disable interrupts to the SBus when set. The default is disabled.
D14	REOD	Reset EODMA , the end-of-DMA interrupt bit in the status register. This value is not stored and need not be reset.
D13	RDINT	Reset the Device Interrupt bit in the status register by setting RDINT to 1. This does not affect the state of the DEVINT signal or the DINT status bit. This value is not stored and need not be reset.
D12	FCLK	Force Clock. Set FCLK to produce the same effect as an external cycle request (DCLK) ; it is not necessary to clear FCLK afterward.
D11–9		Unused

**Table 10. The Command Register**

Bit	S16D_	Description
D8	BCLR	Board Clear. Set BCLR to abort the DMA in process and clear the S16D FIFOs, provided DFRST in the configuration register has not been set to disable a FIFO reset. This operation does not send INIT to the device. This value is not stored and need not be reset.
D7	INIT	Initialize device. The S16D asserts the DEVINIT signal on the interface as long as INIT is set to 1. The DEVINIT signal is not a pulse; the software determines the length of DEVINIT according to the requirements of the device.
D6–4	FNCT3 FNCT2 FNCT1	These function control bits are passed directly to the device interface. They are true when positive: write a 1 to these bits to see a logic high on the cable. When the loopback connector is installed, FNCT2 is connected to the device interrupt input.
D3		Unused
D2	FLUSH	FLUSH is set to 1 when a DMA transfer is aborted and the register files still contain data awaiting a burst transfer. FLUSH causes a final 16-byte transfer to the current DMA buffer. This value is not stored and need not be reset.
D1	DIR	Select the direction of the DMA transfer. DIR is set to 1 when the S16D writes DMA data. When the loopback connector is installed, toggling this bit changes the DEVINT signal.
D0	GO	Initiate the DMA. Clear GO to get ready for the transfer, and pulse GO to initiate the transfer.

**Table 10. The Command Register**

## Status Register

The status register is a 16-bit read-only register at address 0x8000.

Bit	S16D_	Description
D15	INT_S	Shows when ATTN or EODMA interrupts are pending or enabled. INT is set to 1 when the SBus interrupt is asserted. You must clear the DISINT bit of the command register to set INT_S, because INT_S is not available when interrupts are disabled.
D14	EODMA_S	Shows when the end-of-DMA interrupt is pending. EODMA is set when a DMA cycle is complete, ENEOD is set in the configuration register, and SDMAEN is not asserted. Clear EODMA with REOD, bit 14 of the command register. EODMA causes an SBus interrupt if interrupts are enabled—that is, when DISINT (bit 15 of the command register) is 0.
D13	INTDEV_S	Shows when a device interrupt is pending. INTDEV is set to show a pending device interrupt when the external DINT input is asserted (low) and latched, and the ENDINT bit is set high in the configuration register. INTDEV causes an SBus interrupt when DISINT (bit 15 of the command register) is 0, which enables interrupts. Clear ENDINT by setting RDINT, bit 13 of the command register.
D12	0	Always set to 0 on the current S16D.
D11	DINT_S	Reflects the state of the external S16D DINT (Device Interrupt) input. DINT is not latched.
D10–8	STATC_S STATB_S STATA_S	Reflects the state of the external S16D inputs STATA, STATB, and STATC.
D7	DINIT_S	Reads back the inverted state of the command register INIT bit.
D6–4	FNCT3_S FNCT2_S FNCT1_S	Reads back the state of command register bits FNCT1, FNCT2, and FNCT3.
D3	ENEOD_S	End of DMA interrupt enabled; reads back the ones complement of the ENEOD bits in the configuration register.
D2	ENDINT_S	Device interrupt enabled; reads back the ones complement of the ENDINT bits in the configuration register.
D1	DIR_S	Reads back the ones complement of the DIR bit in the command register.
D0	DMAEN_S	Indicates when SBus DMA is in progress. DMAEN is set whenever the SDMAEN signal is asserted (low) showing an active transfer, and is clear when idle.

**Table 11. The Status Register**

## Configuration Register

The configuration register is a 16-bit register at address 0x8002.

Bit	S16D_	Description
D15		Unused
D14	ENEOD	Set ENEOD to 1 to enable the EODMA (end-of-DMA) interrupt. You must clear DISINT, the main interrupt enable from the command register, to enable an SBus interrupt.
D13	ENDINT	Set ENINT to 1 to enable interrupts from the device. You must clear DISINT, the main interrupt enable from the command register, to enable an SBus interrupt.
D12–9		Unused
D8	SWAP	SWAP determines which byte of an SBus half-word ends up on which half of the S16D 16-bit bus. When SWAP is zero, byte swapping is disabled; this is the default—the upper byte of an SBus half word appears on the upper half of the S16D 16-bit bus.
D7–4		Unused
D3	DFRST	Set DFRST to disable a FIFO reset. When set to 1, DFRST prevents BCLR from resetting and emptying the FIFO.
D2	DBMODE	Set DBMODE to disable SBus burst DMA transfers. Use DBMODE for applications in which data must be immediately transferred to memory.
D1	CLKP	CLKP, the data clock polarity bit, determines which edge of the DCLK input initiates the transfer. A 0 on CLKP requires a negative edge to initiate the transfer; this is the default. A 1 on CLKP requires a positive DCLK edge to start a transfer.
		We recommend using the negative edge to initiate transfers, so that unplugging the device cable does not produce a clock edge.
D0	DACKP	DACKP determines the polarity of the DACK (Data Acknowledge) signal. If DACKP is zero, the external DACK is asserted low (negative is true); this is the default. If DACKP is set, the external DACK is asserted high (positive is true).
		If you remove the hardware jumper on the S16D, the DACK polarity is forced to positive true, and DACKP is ignored. Use the jumper for devices which require a positive value to be true DACK from powerup.
		We recommend the negative is true setting for DACK, so that DACK is not asserted when the cable is unplugged from the device.

## DMA Address Register

The DMA address register is a 16- or 32-bit register at address 0x8008.

Bit	Description
A31–A20	These addresses latch the 1 MB page addressed by the DMA.
A19–A1	These address bits contain the starting SBus address of the DMA block. When read, they display the next address to access on the SBus. When DMA is complete, the SBus address equals the next address after DMA End Address.
A0	Set to 0.

**Table 12. The DMA Address Register**

## DMA End Address Register

The DMA address register is a 16- or 32-bit register at address 0x800C.

Bit	Name/Value	Description
D31–D20	0	Set to 0s.
D19–D1	EA[19–1]	The End Address bits specify the last SBus address to transfer in the 1 MB page determined by the DMA Address Register.
D0	0	Set to 0.

**Table 13. The DMA End Address Register**

## Specifications

The S16D conforms to the following specifications.

### SBus Compliance

Number of slots:	1
Transfer size	1, 2, 4, and 16 bytes
DVMA master	Yes
SBus memory	32 bits
Clock rate	16 MHz to 25 MHz

### Device Data Transfer

Format	16-bit parallel word
Handshake	2-wire asynchronous handshake
Transfer types	Input or output
Signal polarity	Data is true. Control signals are programmable; the default is true when zero.
Signals	16 data inputs, 16 data outputs 3 status inputs, 3 control outputs, 1 device interrupt
Buffers	8 KB FIFO for input, 8 KB FIFO for output

### Software

Drivers for Sun OS Version 4.1.x and System V Version 4 (Solaris 2.0)

### Power

5 V at 2 A

### Environmental

Temperature	Operating: 10 to 40° C Nonoperating: -20 to 60° C
Humidity	Operating: 20 to 80% noncondensing at 40° C Nonoperating: 95% noncondensing at 40° C

### Physical

Dimensions	3.3" x 5.78" x 0.5"
Weight	6 oz.

## References

See the UNIX man pages for *ioctl(2)*, *config(8)*, *read(2)*, *open(2)*, *write(2)*, *aioread(2)*, *aiowrite(2)*, and *aiowait(2)* for specific information about these system calls.

See the SunOS Installation documents for more information about configuring the kernel for asynchronous I/O.

## Contacting EDT

For Technical support, device driver updates, sales, or other information, contact us at

**Engineering Design Team, Inc.**

1100 NW Compton, Suite 306

Beaverton, OR 97006

Ph: 503-690-1234

FAX: 503-690-1243

Email: [info@edt.com](mailto:info@edt.com)

WWW: <http://www.edt.com>

## Appendix A `ioctl()` Parameters

Engineering Design Team recommends that applications use the software library interface documented starting on page 7. 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.

**S16G\_STATUS** Get the status register. This parameter is usually used to check STAT lines. Provide an unsigned short as an argument.

**S16x\_CONFIG** Set (**S**) or get (**G**) the configuration register. These `ioctl()` parameters set the configuration bits described in `s16d.h` to match the requirements of the device connected to the S16D. The configuration bits include byte swapping, clock polarity, and data acknowledge polarity. Provide an unsigned short as an argument.

### **EDTx\_RTIMEOUT**

Set (**S**) or get (**G**) the timeout lengths for reads from the S16D. The timeout units are hundredths of seconds. Provide an unsigned short as an argument.

### **EDTx\_WTIMEOUT**

Set (**S**) or get (**G**) the timeout lengths for writes to the S16D. The timeout units are hundredths of seconds. Provide an unsigned short as an argument.

**S16S\_DATA** Set the data register. Setting this parameter puts data on the interface without asserting any handshake signals. Provide an unsigned short as an argument.

**NOTE:** Programmed I/O is much slower than direct memory access for large blocks of data.

**S16G\_DATA** Get the data register. Use this parameter to read the data currently on the S16D input signals. Provide an unsigned short as an argument.

### **S16S\_SEND\_INIT\_PULSE**

Send an INIT pulse. The short value passed is the pulse length in hundredths of a second. Provide an unsigned short as an argument.

### **S16S\_DINT\_SIG**

Set the signal to send to the current process upon receipt of a DINT interrupt. This `ioctl()` parameter enables interrupts so the S16D can receive the attention signal, even when I/O is not in progress. Set the signal to zero to disable the signal upon DINT. Provide an unsigned short as an 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 by EDT for internal purposes. Provide an unsigned short as an 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 an unsigned short as an argument.

**S16G\_DMAADDR, S16G\_DMALAST**

Access the hardware registers. **S16G\_DMAADDR** shows where the DMA is currently accessing the buffer of the user device. These parameters require the address pointed to (the third parameter to `ioctl()`) to contain unsigned integers (32 bits). Provide an unsigned integer as an argument.

**NOTE:** These parameters are provided for completeness—manipulation of the data in these registers should be performed only by the driver. Use the `read` and `write` (or `aioread` and `aiowrite`) system calls to perform DMA; the S16D driver already handles the details of the transfer.

**S16S\_COMMAND**

Write the given word to the command register immediately. Use this as a preliminary handshake, to set up a transfer size with a device and possibly to change FCNT lines. The command register changes with any subsequent `read` or `write` system call. Compare the explanation of the `ioctl()` parameters **S16x\_WRITE\_COMMAND** and **S16x\_READ\_COMMAND** for more information. Provide an unsigned short as an argument.

**S16S\_READ\_COMMAND**

Set the word to write to the command register on subsequent `read` system calls. This word is valid only until the next `open()` of the S16D. Use the **S16S\_DEF\_READ\_COMMAND** parameter to define a word to write to the command register on `reads` that remains valid across `open` calls to the device. Provide an unsigned short as an argument.

**S16G\_READ\_COMMAND**

Get the word that will be written to the command register on subsequent `read` system calls. This word is valid only until the next `open()` of the S16D. Use the **S16S\_DEF\_READ\_COMMAND** parameter to get the word written to the command register on `reads` that remains valid across `open` calls to the device. Provide an unsigned short as an argument.

**S16S\_WRITE\_COMMAND**

Set the word to write to the command register on subsequent `write` system calls. This word is valid only until the next `open()` of the S16D. Use the **S16S\_DEF\_WRITE\_COMMAND** parameter to define a word to write to the command register on `reads` that remains valid across `open` calls to the device. Provide an unsigned short as an argument.

**S16G\_WRITE\_COMMAND**

Get the word to write to the command register on subsequent `write` system calls. This word is valid only until the next `open()` of the S16D. Use the **S16S\_DEF\_WRITE\_COMMAND** parameter to define a word to write to the command register that remains valid across `open` calls to the device. Provide an unsigned short as an argument.

**S16S\_DINT\_ERR**

If **DINT\_SIG** is not enabled, cause a `read` or `write` system call to return with the specified error if the driver receives a `DINT` since the last `read` or `write`. This also produces an error after a transfer, if the `DINT` occurs during the transfer. Provide an unsigned short as an argument.

**EDTS\_DEF\_SHORT\_XFER**

Set short transfer timeout mode. Provide an unsigned short as an argument.

Set **EDTS\_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 **EDTS\_DEF\_SHORT\_XFER** to a number greater than one to reset the timeout counter at the start of each transfer.

If **EDS\_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.

**S16G\_XCOUNT** Get the number of bytes transferred since the start of the current DMA transfer. This parameter requires the address pointed to (the third parameter to *ioctl*) to contain an unsigned integer (32 bits). Provide an unsigned integer as an argument.

**EDTS\_TERMINATE**

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

**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 an unsigned short as an 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.

**EDT\_FREE\_BUF** Turns off continuous ring-buffer mode. No argument is required.

**EDTG\_DONECOUNT**

Use with continuous ring-buffer mode (**S16S\_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 an unsigned integer as an argument.

**EDT6S\_WAKEUP\_DONECOUNT**

Use with continuous ring-buffer mode (**EDTS\_NBUFIO**) and **S16G\_DONECOUNT**. 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.

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.

Provide an unsigned integer as an argument.

**EDTS\_LOCKSTEP**

Use with continuous ring-buffer mode (**EDTS\_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*. Setting the third argument to an address that points to 0 turns off lockstep mode. Provide an unsigned integer as an argument.

**EDTS\_APPBUFS\_COMPLETED**

Use with continuous ring-buffer mode (**EDTS\_NBUFIO**) and lockstep mode (**EDTS\_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 an unsigned integer as an argument.

