

16SDI

**16-bit, 2 to 16 Analog Input Channels
Sigma-Delta ADC, 220K SPS/Ch and 1100K SPS/Ch**

All Form Factors

...-6/16SDI

...-16SDI-HS

...-16HSDI

Linux Device Driver And API Library User Manual

**Manual Revision: October 21, 2024
Driver Release Version 5.7.111.50.0**

**General Standards Corporation
8302A Whitesburg Drive
Huntsville, AL 35802
Phone: (256) 880-8787
Fax: (256) 880-8788**

URL: <http://www.generalstandards.com>

E-mail: sales@generalstandards.com

E-mail: support@generalstandards.com

Preface

Copyright © 2003-2024, **General Standards Corporation**

Additional copies of this manual or other literature may be obtained from:

General Standards Corporation

8302A Whitesburg Dr.

Huntsville, Alabama 35802

Phone: (256) 880-8787

FAX: (256) 880-8788

URL: <http://www.generalstandards.com>

E-mail: sales@generalstandards.com

General Standards Corporation makes no warranty of any kind with regard to this documentation and/or software, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Although extensive editing and reviews are performed before release, **General Standards Corporation** assumes no responsibility for any errors, inaccuracies or omissions herein. This documentation, information and software are made available solely on an “as-is” basis. Nor is there any commitment to update or keep current this documentation.

General Standards Corporation does not assume any liability arising out of the application or use of documentation, software, product or circuit described herein, nor is any license conveyed under any patent rights or any rights of others.

General Standards Corporation assumes no responsibility for any consequences resulting from omissions or errors in this manual or from the use of information contained herein.

General Standards Corporation reserves the right to make any changes, without notice, to this documentation, software or product, to improve accuracy, clarity, reliability, performance, function, or design.

ALL RIGHTS RESERVED.

GSC is a trademark of **General Standards Corporation**.

PLX and PLX Technology are trademarks of PLX Technology, Inc.

Table of Contents

1. Introduction.....	7
1.1. Purpose.....	7
1.2. Acronyms.....	7
1.3. Definitions	7
1.4. Software Overview	7
1.4.1. Basic Software Architecture	7
1.4.2. API Library.....	8
1.4.3. Device Driver	8
1.5. Hardware Overview	8
1.6. Reference Material.....	8
1.7. Licensing.....	9
2. Installation	10
2.1. CPU and Kernel Support.....	10
2.1.1. 32-bit Support Under 64-bit Environments	11
2.2. The /proc/ File System	11
2.3. File List.....	11
2.4. Directory Structure.....	11
2.5. Installation	12
2.6. Removal.....	12
2.7. Overall Make Script.....	12
2.8. Environment Variables	13
2.8.1. GSC_API_COMP_FLAGS.....	13
2.8.2. GSC_API_LINK_FLAGS.....	13
2.8.3. GSC_LIB_COMP_FLAGS.....	13
2.8.4. GSC_LIB_LINK_FLAGS.....	14
2.8.5. GSC_APP_COMP_FLAGS.....	14
2.8.6. GSC_APP_LINK_FLAGS.....	14
3. Main Interface Files.....	15
3.1. Main Header File	15
3.2. Main Library File.....	15
3.2.1. Build	15
3.2.2. System Libraries.....	16
3.2.3. Shared Object Script: Build the Main Libraries as Shared Object Files.....	16
4. API Library	17
4.1. Files.....	17
4.2. Build	17
4.3. Library Use	17

4.4. Macros	17
4.4.1. IOCTL Services	18
4.4.2. Registers	18
4.5. Data Types	18
4.6. Functions	19
4.6.1. sdi_close()	19
4.6.2. sdi_init()	19
4.6.3. sdi_ioctl()	20
4.6.4. sdi_open()	21
4.6.5. sdi_read()	22
4.7. IOCTL Services	23
4.7.1. SDI_IOCTL_AIN_BUF_CLEAR	23
4.7.2. SDI_IOCTL_AIN_BUF_INPUT	23
4.7.3. SDI_IOCTL_AIN_BUF_THRESH	23
4.7.4. SDI_IOCTL_AIN_BUF_THRESH_STS	24
4.7.5. SDI_IOCTL_AIN_MODE	24
4.7.6. SDI_IOCTL_AIN_RANGE	24
4.7.7. SDI_IOCTL_AUTOCAL	25
4.7.8. SDI_IOCTL_AUTOCAL_STATUS	25
4.7.9. SDI_IOCTL_CH_GRP_X_SRC	25
4.7.10. SDI_IOCTL_CHANNEL_ORDER	26
4.7.11. SDI_IOCTL_CHANNELS_READY	26
4.7.12. SDI_IOCTL_DATA_FORMAT	27
4.7.13. SDI_IOCTL_INIT_MODE	27
4.7.14. SDI_IOCTL_INITIALIZE	27
4.7.15. SDI_IOCTL_IRQ_SEL	28
4.7.16. SDI_IOCTL_QUERY	28
4.7.17. SDI_IOCTL_REG_MOD	29
4.7.18. SDI_IOCTL_REG_READ	30
4.7.19. SDI_IOCTL_REG_WRITE	30
4.7.20. SDI_IOCTL_RX_IO_ABORT	31
4.7.21. SDI_IOCTL_RX_IO_MODE	31
4.7.22. SDI_IOCTL_RX_IO_TIMEOUT	32
4.7.23. SDI_IOCTL_RATE_DIV_XX_NDIV	32
4.7.24. SDI_IOCTL_RATE_GEN_X_NRATE	32
4.7.25. SDI_IOCTL_SW_SYNC	33
4.7.26. SDI_IOCTL_SW_SYNC_MODE	33
4.7.27. SDI_IOCTL_WAIT_CANCEL	33
4.7.28. SDI_IOCTL_WAIT_EVENT	34
4.7.29. SDI_IOCTL_WAIT_STATUS	36
5. The Driver	38
5.1. Files	38
5.2. Build	38
5.3. Startup	38
5.3.1. Manual Driver Startup Procedures	38
5.3.2. Automatic Driver Startup Procedures	39
5.4. Verification	40
5.5. Version	41
5.6. Shutdown	41

6. Document Source Code Examples.....	42
6.1. Files.....	42
6.2. Build	42
6.3. Library Use	42
7. Utilities Source Code.....	43
7.1. Files.....	43
7.2. Build	43
7.3. Library Use	43
8. Operating Information	44
8.1. Debugging Aids	44
8.1.1. Device Identification	44
8.1.2. Detailed Register Dump	44
8.2. Analog Input Configuration	44
8.3. Data Transfer Modes.....	44
8.3.1. PIO - Programmed I/O	45
8.3.2. BMDMA - Block Mode DMA	45
8.3.3. DMDMA - Demand Mode DMA	45
8.4. Multi-Board Synchronization	45
8.4.1. Star Configuration	45
8.4.2. Daisy Chain Configuration	46
9. Sample Applications	47
9.1. billion - Billion Byte Read - ../billion/	47
9.2. clock - Clock Output - ../clock/	47
9.3. fsamp - Sample Rate - ../fsamp/	47
9.4. id - Identify Board - ../id/	47
9.5. regs - Register Access - ../regs/	47
9.6. rxrate - Receive Rate - ../rxrate/	47
9.7. savedata - Save Acquired Data - ../savedata/	47
9.8. sbtest - Single Board Test - ../sbtest/	47
9.9. sw_sync - Software Sync - ../sw_sync/	47
Document History	48

Table of Figures

Figure 1 Basic architectural representation.....	8
Figure 2 The <i>star</i> configuration with four or more boards requires a Clock Driver board.	46
Figure 3 The <i>star</i> configuration with only two or three boards does not require a Clock Driver board.	46
Figure 4 In this configuration the clock and sync signals are daisy chained from one board to the next.	46

1. Introduction

1.1. Purpose

The purpose of this document is to describe the interface to the 16SDI API Library and to the underlying Linux device driver. The API Library software provides the interface between "Application Software" and the device driver. The driver software provides the interface between the API Library and the actual 16SDI hardware. The API Library and driver interfaces are based on the board's functionality.

1.2. Acronyms

The following is a list of commonly occurring acronyms which may appear throughout this document.

Acronyms	Description
ADC	Analog-to-Digital Converter
API	Application Programming Interface
BMDMA	Block Mode DMA
DMA	Direct Memory Access
DMDMA	Demand Mode DMA
GSC	General Standards Corporation
PC104P	This refers to the PC/104+ form factor.
PCI	Peripheral Component Interconnect
PCIe	PCI Express
PIO	Programmed I/O
PMC	PCI Mezzanine Card

1.3. Definitions

The following is a list of commonly occurring terms which may appear throughout this document.

Term	Definition
...	This is a shortcut representation of the 16SDI installation directory or any of its subdirectories.
16SDI	This is used as a general reference to any device supported by this driver.
API Library	This is a library that provides application-level access to 16SDI hardware.
Application	This is a user mode process, which runs in user space with user mode privileges.
Driver	This is the 16SDI device driver, which runs in kernel space with kernel mode privileges.
Library	This is usually a general reference to the API Library.

1.4. Software Overview

1.4.1. Basic Software Architecture

This section describes the general architecture for the basic components that comprise 16SDI applications. The overall architecture is illustrated in Figure 1 below.

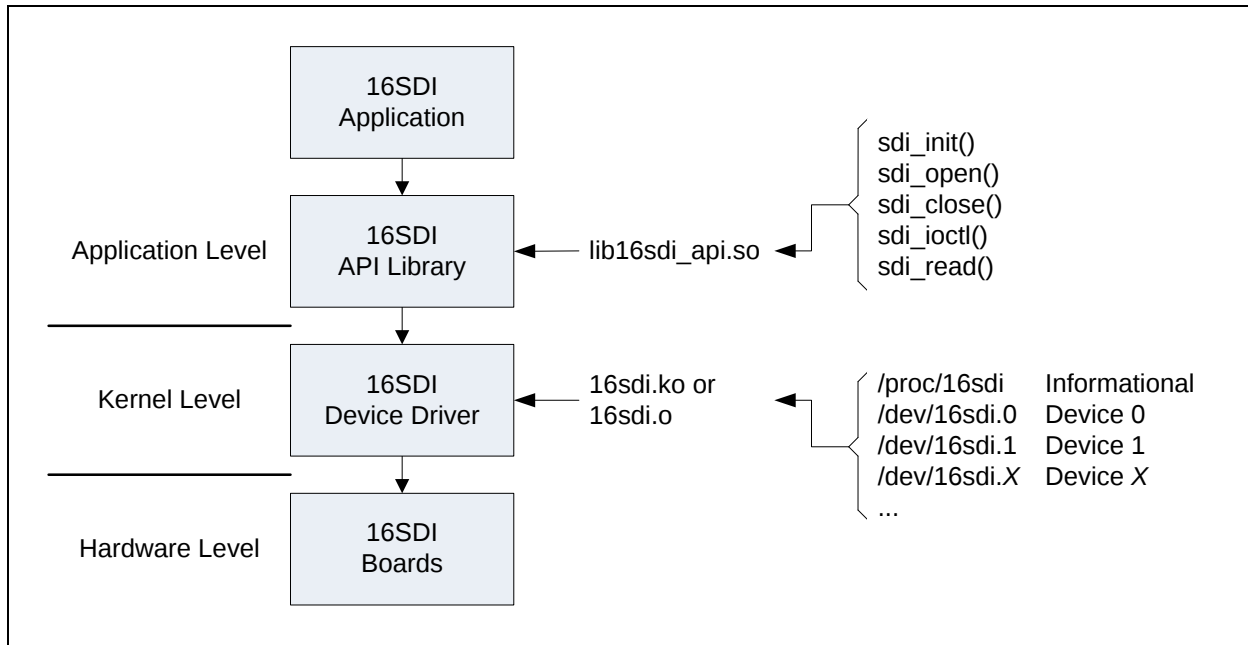


Figure 1 Basic architectural representation.

1.4.2. API Library

The primary means of accessing 16SDI boards is via the 16SDI API Library. This library forms a layer between the application and the driver. Additional information is given in section 3.2.3 (page 16). With the library, applications are able to open and close a device and, while open, perform I/O control and read operations.

1.4.3. Device Driver

The device driver is the host software that provides a means of communicating directly with 16SDI hardware. The driver executes under control of the operating system and runs in Kernel Mode as a Kernel Mode device driver. The driver is implemented as a standard dynamically loadable Linux device driver written in the C programming language. While applications can access the driver directly without use of the API Library, it is recommended that all access is made through the library.

1.5. Hardware Overview

The 16SDI boards are high-performance multi-channel, 16-bit analog input board. The number of input channels is from two to 16, depending on the model and options ordered. The host side connection is PCI based and the form factor is according to the model ordered. The base models are capable of acquiring data at up to 220K samples per second over each channel. The high-speed models are capable of acquiring data at up to 1100K samples per second over each channel. Internal clocking permits sampling rates down to 5K S/S for the base models and 30K S/S for the high-speed models. Onboard storage permits data buffering of either 64K samples or 256K samples, depending on the model ordered, for all channels collectively, between the cable interface and the PCI bus. This allows a 16SDI to sustain continuous throughput from the cable interface independent of the PCI bus interface. The 16SDI also permits multiple boards to be synchronized so that all boards sample data in unison. In addition, the board includes autocalibration capability.

1.6. Reference Material

The following reference material may be of particular benefit in using the 16SDI. The specifications provide the information necessary for an in depth understanding of the specialized features implemented on this board.

- The applicable *16SDI User Manual* from General Standards Corporation.
- The PCI9080 PCI Bus Master Interface Chip data handbook from PLX Technology, Inc.

PLX Technology Inc.
870 Maude Avenue
Sunnyvale, California 94085 USA
Phone: 1-800-759-3735
WEB: <http://www.plxtech.com>

1.7. Licensing

For licensing information please refer to the text file `LICENSE.txt` in the root installation directory.

2. Installation

2.1. CPU and Kernel Support

The driver is designed to operate with Linux kernel versions 6.x, 5.x, 4.x, 3.x, 2.6, 2.4 and 2.2 running on a PC system with one or more x86 processors. This release of the driver supports the below listed kernels.

Kernel	Distribution
6.8.5	Red Hat Fedora Core 40
6.5.6	Red Hat Fedora Core 39
6.2.9	Red Hat Fedora Core 38
6.0.7	Red Hat Fedora Core 37
5.17.5	Red Hat Fedora Core 36
5.14.10	Red Hat Fedora Core 35
5.11.12	Red Hat Fedora Core 34
5.8.15	Red Hat Fedora Core 33
5.6.6	Red Hat Fedora Core 32
5.3.7	Red Hat Fedora Core 31
5.0.9	Red Hat Fedora Core 30
4.18.16	Red Hat Fedora Core 29
4.16.3	Red Hat Fedora Core 28
4.13.9	Red Hat Fedora Core 27
4.11.8	Red Hat Fedora Core 26
4.8.6	Red Hat Fedora Core 25
4.5.5	Red Hat Fedora Core 24
4.2.3	Red Hat Fedora Core 23
4.0.4	Red Hat Fedora Core 22
3.17.4	Red Hat Fedora Core 21
3.11.10	Red Hat Fedora Core 20
3.9.5	Red Hat Fedora Core 19
3.6.10	Red Hat Fedora Core 18
3.3.4	Red Hat Fedora Core 17
3.1.0	Red Hat Fedora Core 16
2.6.38	Red Hat Fedora Core 15
2.6.35	Red Hat Fedora Core 14
2.6.33	Red Hat Fedora Core 13
2.6.31	Red Hat Fedora Core 12
2.6.29	Red Hat Fedora Core 11
2.6.27	Red Hat Fedora Core 10
2.6.25	Red Hat Fedora Core 9
2.6.23	Red Hat Fedora Core 8
2.6.21	Red Hat Fedora Core 7
2.6.18	Red Hat Fedora Core 6
2.6.15	Red Hat Fedora Core 5
2.6.11	Red Hat Fedora Core 4
2.6.9	Red Hat Fedora Core 3
2.4.18	Red Hat 8.0

NOTE: Some older kernel versions are supported (the sources are maintained), but are not tested.

NOTE: While only Red Hat Fedora distributions are listed, numerous other distributions are supported and have been tested on an as needed basis.

NOTE: The driver will have to be built before being used as it is provided in source form only.

NOTE: The driver has not been tested with a non-versioned kernel.

NOTE: The driver is designed for SMP support, but has not undergone SMP specific testing.

2.1.1. 32-bit Support Under 64-bit Environments

This driver supports 32-bit applications under 64-bit environments. The availability of this feature in the kernel depends on a 64-bit kernel being configured to support 32-bit application compatibility. Additionally, 2.6 kernels prior to 2.6.11 implemented 32-bit compatibility in a way that resulted in some drivers not being able to take advantage of the feature. (In these kernels a driver's IOCTL command codes must be globally unique. Beginning with 2.6.11 this requirement has been lifted.) If the driver is not able to provide 32-bit support under a 64-bit kernel, the "32-bit support" field in the `/proc/16sdi` file will be "no".

2.2. The `/proc/` File System

While the driver is running, the text file `/proc/16sdi` can be read to obtain information about the driver and the boards it detects. Each file entry includes an entry name followed immediately by a colon, a space character, and the entry value. Below is an example of what appears in the file, followed by descriptions of each entry.

```
version: 5.7.111.50
32-bit support: yes
boards: 1
models: 16SDI
```

Entry	Description
version	This gives the driver version number in the form <code>x.x.x.x</code> .
32-bit support	This reports the driver's support for 32-bit applications. This will be either "yes" or "no" for 64-bit driver builds and "yes (native)" for 32-bit builds.
boards	This identifies the total number of boards the driver detected.
models	This gives a comma separated list of the basic model number for each board the driver detected. The model numbers are listed in the same order that the boards are accessed via the API Library's open function.

2.3. File List

This release consists of the below listed primary files. The archive content is described in following subsections.

File	Description
<code>16sdi.linux.tar.gz</code>	This archive contains the driver, the API Library and all related files.
<code>16sdi_linux_um.pdf</code>	This is a PDF version of this user manual, which is included in the archive.

2.4. Directory Structure

The following table describes the directory structure utilized by the installed files. During installation the directory structure is created and populated with the respective files.

Directory	Description
<code>16sdi/</code>	This is the driver root directory. It contains the documentation, the Overall Make Script (section 2.7, page 12) and the below listed subdirectories.
<code>.../api/</code>	This directory contains the API Library source files (section 3.2.3, page 16).
<code>.../docsrc/</code>	This directory contains the source files for the code samples given in this document (section 6, page 42).

.../driver/	This directory contains the device driver source files (section 5, page 38).
.../include/	This directory contains the header files for the various libraries.
.../lib/	This directory contains all of the libraries built from the installed sources.
.../samples/	This directory contains the sample application subdirectories and all of their corresponding source files (section 9, page 47).
.../utils/	This directory contains the source files for the utility libraries used by the sample applications (section 7, page 43).

2.5. Installation

Perform installation following the below listed steps. This installs the device driver, the API Library and all related sources and documentation.

1. Create and change to the directory where the files are to be installed, such as `/usr/src/linux/drivers/`. (The path name may vary among distributions and kernel versions.)
2. Copy the archive file `16sdi.linux.tar.gz` into the current directory.
3. Issue the following command to decompress and extract the files from the provided archive. This creates the directory `16sdi` in the current directory, and then copies all of the archive's files into this new directory.

```
tar -xzvf 16sdi.linux.tar.gz
```

2.6. Removal

Perform removal following the below listed steps. This removes the device driver, the API Library and all related sources and documentation.

NOTE: The following steps may require elevated privileges.

1. Shutdown the driver as described in section 5.6 (page 41).
2. Change to the directory where the driver archive was installed, which may have been `/usr/src/linux/drivers/`. (The path name may vary among distributions and kernel versions.)
3. Issue the below command to remove the driver archive and all of the installed driver files.

```
rm -rf 16sdi.linux.tar.gz 16sdi
```

4. Issue the below command to remove all of the installed device nodes.

```
rm -f /dev/16sdi.*
```

5. If the automatic startup procedure was adopted (section 5.3.2, page 39), then edit the system startup script `rc.local` and remove the line that invokes the 16SDI's start script. The file `rc.local` should be located in the `/etc/rc.d/` directory.

2.7. Overall Make Script

An Overall Make Script is included in the root installation directory. Executing this script will perform a make for all build targets included in the release. The script also loads the driver and copies the API Library to `/usr/lib/`. The script is named `make_all`. Follow the below steps to perform an overall make and to load the driver.

NOTE: The following steps may require elevated privileges.

1. Change to the driver root directory (`.../16sdi/`).
2. Remove existing build targets using the below command. This does not unload the driver.

```
./make_all clean
```

3. Issue the following command to make all archive targets and to load the driver.

```
./make_all
```

2.8. Environment Variables

Some build environments may require compiler or linker options not present in the provided make files. To accommodate local environment specific requirements, the provided make files incorporate support for the following set of GSC specific environment variables.

2.8.1. GSC_API_COMP_FLAGS

This environment variable accommodates adding compiler command line options when compiling source files for the API Library. The compiler used by the API Library make file is “gcc”. The content of this environment variable is noted in the make file’s output to the screen. The table below shows a portion of the screen output. The “xxx” in the table refers to the contents of the environment variable. This environment variable has no effect on compiling any other distributed source files or linking of any object files.

Undefined or Empty	== Compiling: init.c == Compiling: ioctl.c == Compiling: open.c
Defined and Not Empty	== Compiling: init.c (added 'xxx') == Compiling: ioctl.c (added 'xxx') == Compiling: open.c (added 'xxx')

2.8.2. GSC_API_LINK_FLAGS

This environment variable accommodates adding linker command line options when linking object files for the API Library. The linker used by the API Library make file is “ld”. The content of this environment variable is noted in the make file’s output to the screen. The table below shows a portion of the screen output. The “xxx” in the table refers to the contents of the environment variable. This environment variable has no effect on compiling of any source files or linking of any other object files.

Undefined or Empty	==== Linking: ../lib/lib16sdi_api.so
Defined and Not Empty	==== Linking: ../lib/lib16sdi_api.so (added 'xxx')

2.8.3. GSC_LIB_COMP_FLAGS

This environment variable accommodates adding compiler command line options when compiling source files for the utility libraries. The compiler used by the utility library make files is “gcc”. The content of this environment variable is noted in the make files’ output to the screen. The table below shows a portion of the screen output. The “xxx” in the table refers to the contents of the environment variable. This environment variable has no effect on compiling any other distributed source files or linking of any object files.

Undefined or Empty	== Compiling: close.c == Compiling: init.c == Compiling: ioctl.c
Defined and Not Empty	== Compiling: close.c (added 'xxx') == Compiling: init.c (added 'xxx') == Compiling: ioctl.c (added 'xxx')

2.8.4. GSC_LIB_LINK_FLAGS

This environment variable accommodates adding linker command line options when linking object files for the utility libraries. The linker used by the utility library make files is “ld”. The content of this environment variable is noted in the make files’ output to the screen. The table below shows a portion of the screen output. The “xxx” in the table refers to the contents of the environment variable. This environment variable has no effect on compiling of any source files or linking of any other object files.

Undefined or Empty	==== Linking: ../lib/16sdi_utils.a
Defined and Not Empty	==== Linking: ../lib/16sdi_utils.a (added 'xxx')

2.8.5. GSC_APP_COMP_FLAGS

This environment variable accommodates adding compiler command line options when compiling source files for the sample applications. The compiler used by the sample application make files is “gcc”. The content of this environment variable is noted in the make files’ output to the screen. The table below shows a portion of the screen output. The “xxx” in the table refers to the contents of the environment variable. This environment variable has no effect on compiling any other distributed source files or linking of any object files.

Undefined or Empty	== Compiling: main.c == Compiling: perform.c
Defined and Not Empty	== Compiling: main.c (added 'xxx') == Compiling: perform.c (added 'xxx')

2.8.6. GSC_APP_LINK_FLAGS

This environment variable accommodates adding linker command line options when linking object files for the sample applications. The linker used by the sample application make files is “gcc”. The content of this environment variable is noted in the make files’ output to the screen. The table below shows a portion of the screen output. The “xxx” in the table refers to the contents of the environment variable. This environment variable has no effect on compiling of any source files or linking of any other object files.

Undefined or Empty	==== Linking: id
Defined and Not Empty	==== Linking: id (added 'xxx')

3. Main Interface Files

This section gives general information on the suggested device interface files to use when developing 16SDI based applications.

3.1. Main Header File

Throughout the remainder of this document references are made to various header files included as part of the 16SDI driver installation. For ease of use it is suggested that applications include only the single header file shown below rather than individually including those headers identified separately later in this document. Including this header file pulls in all other pertinent 16SDI specific header files. Therefore, sources may include only this one 16SDI header and make files may reference only this one 16SDI include directory.

Description	File	Location
Header File	16sdi_main.h	.../include/

3.2. Main Library File

Throughout the remainder of this document references are made to various statically linkable libraries included as part of the 16SDI driver installation. For ease of use it is suggested that applications link only the single library file shown below rather than individually linking those libraries identified separately later in this document. Linking this library file pulls in all other static libraries included with the driver. Therefore, make files may reference only this one 16SDI static library and only this one 16SDI library directory.

Description	File	Location
Static Library	16sdi_main.a	.../lib/
	16sdi_multi.a	

NOTE: For applications using the 16SDI and no other GSC devices, link the 16sdi_main.a library. For applications using multiple GSC device types, link the xxxx_main.a library for one of the devices and the xxxx_multi.a library for the others. Linking multiple xxxx_main.a libraries may likely produce link errors due to duplicate symbols being defined. While it may make little or no difference, it is recommended that one choose the xxxx_main.a library from the driver with the largest number in positions three (x.x.X.x.x) and/or four (x.x.x.X.x) in the driver release version number.

NOTE: The 16SDI API Library is implemented as a shared library and is thus not linked with the 16SDI Main Library. The API Library must be linked with applications by adding the argument -l16sdi_api to the linker command line.

3.2.1. Build

The main library is built via the Overall Make Script (section 2.7, page 12). However, the main library can be built separately following the below steps.

1. Change to the directory where the main library resides (.../lib/).
2. Remove existing build targets using the below command.

```
make clean
```

3. Build the main library by issuing the below command.

```
make
```

3.2.2. System Libraries

In addition to linking the static library named above, as well as the API Library shared object file, applications may need to also link in additional system libraries as noted below.

Library	gcc Link Flag
Math	-lm
POSIX Thread	-lpthread
Real Time	-lrt

3.2.3. Shared Object Script: Build the Main Libraries as Shared Object Files

The main libraries built via the Overall Make Script (section 2.7, page 12) are static library files. Some applications however, require that the Main Libraries be accessed as shared object files. Generating shared object files require that all of the static libraries be recompiled for this purpose and linked as .so files. This is done using the Shared Object Script named below. When run, the script invokes the Overall Make Script to clean all existing build targets, deletes the two shared object files named below, if they exist, defines an environment variable used by all of the static library make files, invokes the Overall Make Script again to rebuild all existing build targets then invokes make on the library make file (.../lib/makefile) to link the shared object files. The required manual steps are as follows.

1. Change to the directory where the main library files reside (.../lib/).
2. Execute the below script.

```
./static_to_shared.sh
```

Running the above-named Shared Object Script produces the files given in the table below. These shared object files fulfill the same purpose as the similarly named static libraries as described in the note under section 3.2 above. Refer to that note when selecting which shared object file to use.

Description	File	Location
Shared Object Files	lib16sdi_main.so	.../lib/
	lib16sdi_multi.so	
	lib16sdi_all.so†	

† This library includes all generated libraries, including the API Library shared object file content.

The shared object files can be linked via two different methods. In the first method, the application linker command line can explicitly name the file in the same manner as is done were it a static library. This is the method used by the sample applications, all of which use the 16SDI API Library, which itself is a shared object file. This file is also found in the .../lib/ subdirectory. In the second method, the .so files are copied to the /usr/lib/ subdirectory and are referenced on the application's linker command line as given in the table below.

Library	gcc Link Flag
lib16sdi_main.so	-l16sdi_main
lib16sdi_multi.so	-l16sdi_multi
lib16sdi_all.so†	-l16sdi_all

† This library includes all generated libraries, including the API Library shared object file content.

4. API Library

The 16SDI API Library is the software interface between user applications and the 16SDI device driver. The interface is accessed by including the header file `16sdi_api.h`.

NOTE: Contact General Standards Corporation if additional library functionality is required.

4.1. Files

The library files are summarized in the table below.

Description	File	Location
Source Files	*.c, *.h ...	.../api/
Header File	<code>16sdi_api.h</code>	.../include/
Library File	<code>lib16sdi_api.so</code>	.../lib/ /usr/lib/ †

† The shared object library is automatically copied to `/usr/lib/` when it is built.

4.2. Build

The API Library is built via the Overall Make Script (section 2.7, page 12), but can be built separately following the below steps.

NOTE: The following steps may require elevated privileges.

1. Change to the directory where the library sources are installed (`.../api/`).
2. Remove existing build targets using the below command.

```
make clean
```

3. Compile the source files and build the library by issuing the below command. This step copies the API Library file to `/usr/lib/`.

```
make
```

4.3. Library Use

The library is used at application compile time, at application link time and at application run time. At compile time include the below listed header file in each source file using a component of the Library interface. Also, edit the include file search path to locate the header file in the below listed directory. At link time the Library's shared object file is linked via the linker command line. This can be done by naming the `.so` file explicitly or by adding the below linker command line argument. At run time the library is found in the directory `/usr/lib/`. (The shared object file is automatically copied to `/usr/lib/` when it is built.)

Description	File	Location	Linker Argument
Header File	<code>16sdi_api.h</code>	.../include/	
Shared Object Library	<code>lib16sdi_api.so</code>	.../lib/ /usr/lib/	<code>-l16sdi_api</code>

4.4. Macros

The API Library and driver interfaces include the following macros, which are defined in `16sdi.h`.

4.4.1. IOCTL Services

The IOCTL macros are documented in section 4.7 (page 23).

4.4.2. Registers

The following gives the complete set of 16SDI registers.

4.4.2.1. GSC Registers

The following table gives the complete set of GSC specific 16SDI registers. Please note that the set of registers supported by any given device may vary according to model and firmware version. For the set of supported registers and their detailed definitions refer to the appropriate *16SDI User Manual*.

NOTE: Refer to the output of the “id” sample application (.../id/) for a complete list of the registers supported by the device being accessed.

Macros	Description
SDI_GSC_AVR	Autocal Values Register (AVR)
SDI_GSC_BCR	Board Control Register (BCR)
SDI_GSC_BRR	Board Revision Register (BRR)
SDI_GSC_BSR	Buffer Size Register (BSR)
SDI_GSC_BTR	Buffer Threshold Register (BTR)
SDI_GSC_IDBR	Input Data Buffer Register (IDBR)
SDI_GSC_RAR	Rate Assignments Register (RAR)
SDI_GSC_RCAR	Rate Control A Register (RCAR)
SDI_GSC_RCBR	Rate Control B Register (RXBR)
SDI_GSC_RCCR	Rate Control C Register (RCCR)
SDI_GSC_RCDR	Rate Control D Register (RCDR)
SDI_GSC_RD01R	Rate Divisor 00/01 Register (RD01R)
SDI_GSC_RD23R	Rate Divisor 02/03 Register (RD23R)
SDI_GSC_RD45R	Rate Divisor 04/05 Register (RD45R)
SDI_GSC_RD67R	Rate Divisor 06/07 Register (RD67R)
SDI_GSC_RD89R	Rate Divisor 08/09 Register (RD89R)
SDI_GSC_RDABR	Rate Divisor 10/11 Register (RDABR)
SDI_GSC_RDCDR	Rate Divisor 12/13 Register (RDCDR)
SDI_GSC_RDEFR	Rate Divisor 14/15 Register (RDEFR)

4.4.2.2. PCI Configuration Registers

Access to the PCI registers is seldom required so these registers are not listed here. For the complete list of the PCI register identifiers refer to the driver header file `gsc_pci9080.h`, which is automatically included via `16sdi_api.h`.

4.4.2.3. PLX Feature Set Registers

Access to the PLX registers is seldom required so these registers are not listed here. For the complete list of the PLX register identifiers refer to the driver header file `gsc_pci9080.h`, which is automatically included via `16sdi_api.h`.

4.5. Data Types

The data types used by the API Library are described with the IOCTL services with which they are used. For additional information refer to section 4.7 (page 23).

4.6. Functions

The interface includes the following functions. The return values reflect the completion status of the requested operation. A return value less than zero always reflects an error condition. The table below summarizes the error status values. For the I/O function, read, non-negative return values reflect the number of bytes transferred between the application and the interface. A value equal to the requested transfer size indicates complete success. Return values less than the requested transfer size indicate that the I/O timeout expired. For the other API function calls a return value of zero indicates success.

Return Value	Description
< 0	This is the value “(-errno)” (see <code>errno.h</code>).

4.6.1. `sdi_close()`

This function is the entry point to close a connection made via the API's open call (section 4.6.4, page 21). The device is put in an initialized state before this call returns.

Prototype

```
int sdi_close(int fd);
```

Argument	Description
fd	This is the file descriptor obtained from the open service (section 4.6.4, page 21).

Return Value	Description
0	The operation succeeded.
< 0	An error occurred. See error value description above.

Example

```
#include <stdio.h>

#include "16sdi_dsl.h"

int sdi_close_dsl(int fd)
{
    int errs;
    int ret;

    ret = sdi_close(fd);

    if (ret)
        printf("ERROR: sdi_close() returned %d\n", ret);

    errs = ret ? 1 : 0;
    return(errs);
}
```

4.6.2. `sdi_init()`

This function is the entry point to initializing the 16SDI API Library and must be the first call into the Library. This function may be called more than once, but only the first successful call actually initializes the library. Subsequent calls perform no operation at all. All other API calls return a failure status when the API Library is not initialized.

Prototype

```
int sdi_init(void);
```

Return Value	Description
0	The operation succeeded.
< 0	An error occurred. See error value description above.

Example

```
#include <stdio.h>

#include "16sdi_dsl.h"

int sdi_init_dsl(void)
{
    int errs;
    int ret;

    ret = sdi_init();

    if (ret)
        printf("ERROR: sdi_init() returned %d\n", ret);

    errs = ret ? 1 : 0;
    return(errs);
}
```

4.6.3. sdi_ioctl()

This function is the entry point to performing setup and control operations on a 16SDI. This function should only be called after a successful open of the respective device. The specific operation performed varies according to the `request` argument. The `request` argument also governs the use and interpretation of the `arg` argument. The set of supported options for the `request` argument consists of the IOCTL services supported by the driver, which are defined in section 4.7 (page 23).

Prototype

```
int sdi_ioctl(int fd, int request, void* arg);
```

Argument	Description
<code>fd</code>	This is the file descriptor obtained from the open service (section 4.6.4, page 21).
<code>request</code>	This specifies the desired operation to be performed (section 4.7, page 23).
<code>arg</code>	This is specific to the IOCTL operation specified by the <code>request</code> argument.

Return Value	Description
0	The operation succeeded.
< 0	An error occurred. See error value description above.

Example

```
#include <stdio.h>

#include "16sdi_dsl.h"
```

```

int sdi_ioctl_dsl(int fd, int request, void* arg)
{
    int errs;
    int ret;

    ret = sdi_ioctl(fd, request, arg);

    if (ret)
        printf("ERROR: sdi_ioctl() returned %d\n", ret);

    errs = ret ? 1 : 0;
    return(errs);
}

```

4.6.4. sdi_open()

This function is the entry point to open a connection to a 16SDI board. Before returning, the initialize IOCTL service is called to reset all hardware and software settings to their defaults.

Prototype

```
int sdi_open(int device, int share, int* fd);
```

Argument	Description						
device	This is the zero-based index of the 16SDI to access. †						
share	Open the device in Shared Access Mode? If non-zero the device is opened in Shared Access Mode (see below). If zero the device is opened in Exclusive Access Mode (see below).						
fd	The device handle is returned here. The pointer cannot be NULL. Values returned are as follows. <table> <tr> <th>Value</th><th>Description</th></tr> <tr> <td>>= 0</td><td>This is the handle to use to access the device in subsequent calls.</td></tr> <tr> <td>-1</td><td>There was an error. The device is not accessible.</td></tr> </table>	Value	Description	>= 0	This is the handle to use to access the device in subsequent calls.	-1	There was an error. The device is not accessible.
Value	Description						
>= 0	This is the handle to use to access the device in subsequent calls.						
-1	There was an error. The device is not accessible.						

† The index value -1 can also be given to acquire driver information (section 2.2, page 11).

Return Value	Description
0	The operation succeeded.
< 0	An error occurred. See error value description above.

Example

```

#include <stdio.h>

#include "16sdi_dsl.h"

int sdi_open_dsl(int device, int share, int* fd)
{
    int errs;
    int ret;

    ret = sdi_open(device, share, fd);
}

```

```

    if (ret)
        printf("ERROR: sdi_open() returned %d\n", ret);

    errs    = ret ? 1 : 0;
    return(errs);
}

```

4.6.4.1. Access Modes

The value of the `share` argument determines the device access mode, as follows.

Shared Access Mode:

Shared Access Mode allows multiple applications to access the same device simultaneously. In this mode, the first successful open request returns with the device in an initialized state. Subsequent successful Shared Access Mode open requests do not affect the state of the device. Once opened in Shared Access Mode, the device access remains in this mode until all Shared Access Mode accesses release the device with a close request.

Exclusive Access Mode:

Exclusive Access Mode allows a single application to acquire exclusive access to a device. In this mode, a successful open request returns with the device in an initialized state. While open in this mode all subsequent open requests will fail regardless of the access mode requested. Once opened in Exclusive Access Mode, the device access remains in this mode until released by the application with a close request.

4.6.5. `sdi_read()`

This function is the entry point to reading data from an open connection. The function reads up to `bytes` bytes.

NOTE: If an open was performed using an index of `-1`, then read requests will acquire information from the driver (section 2.2, page 11) rather than data from a device.

NOTE: For additional information refer to the Data Transfer Modes section (section 8.3, page 44).

Prototype

```
int sdi_read(int fd, void* dst, size_t bytes);
```

Argument	Description
<code>fd</code>	This is the file descriptor obtained from the open service (section 4.6.4, page 21).
<code>dst</code>	The data read is put here.
<code>bytes</code>	This is the desired number of bytes to read. This must be a multiple of four (4).

Return Value	Description
0 to <code>bytes</code>	The operation succeeded. When reading from a device, a value less than <code>bytes</code> indicates that the I/O timeout period lapsed (section 4.7.22, page 32) before the entire request could be satisfied.
< 0	An error occurred. See error value description above.

Example

```
#include <stdio.h>
```

```
#include "16sdi_dsl.h"

int sdi_read_dsl(int fd, void* dst, size_t bytes, size_t* qty)
{
    int errs;
    int ret;

    ret = sdi_read(fd, dst, bytes);

    if (ret < 0)
        printf("ERROR: sdi_read() returned %d\n", ret);

    if (qty)
        qty[0] = (ret < 0) ? 0 : (size_t) ret;

    errs = (ret < 0) ? 1 : 0;
    return(errs);
}
```

4.7. IOCTL Services

The 16SDI API Library and device driver implement the following IOCTL services. Each service is described along with the applicable `sdi_ioctl()` function arguments.

4.7.1. SDI_IOCTL_AIN_BUF_CLEAR

This service clears the Analog Input Buffer of its content.

Usage

Argument	Description
request	SDI_IOCTL_AIN_BUF_CLEAR
arg	Not used.

4.7.2. SDI_IOCTL_AIN_BUF_INPUT

This service enables or disables input into the Analog Input Buffer.

Usage

Argument	Description
request	SDI_IOCTL_AIN_BUF_INPUT
arg	s32*

Valid argument values are as follows.

Value	Description
-1	This option returns the current setting.
SDI_AIN_BUF_INPUT_DISABLE	This option disables input to the Analog Input Buffer.
SDI_AIN_BUF_INPUT_ENABLE	This option enables input to the Analog Input Buffer.

4.7.3. SDI_IOCTL_AIN_BUF_THRESH

This service sets the threshold level setting for the Analog Input Buffer.

Usage

Argument	Description
request	SDI_IOCTL_AIN_BUF_THRESH
arg	s32*

Valid settings are from zero to the size of the Analog Input Buffer, which varies among the board models.

4.7.4. SDI_IOCTL_AIN_BUF_THRESH_STS

This service reports the status of the input buffer threshold flag.

Usage

Argument	Description
request	SDI_IOCTL_AIN_BUF_THRESH_STS
arg	s32*

Valid argument values returned are as follows.

Value	Description
SDI_AIN_BUF_THRESH_STS_CLEAR	The input buffer fill level is equal to or less than the programmed threshold level.
SDI_AIN_BUF_THRESH_STS_SET	The input buffer fill level is greater than the programmed threshold level.

4.7.5. SDI_IOCTL_AIN_MODE

This service sets to Analog Input Mode.

Usage

Argument	Description
request	SDI_IOCTL_AIN_MODE
arg	s32*

Valid argument options are as follows.

Value	Description
-1	This returns the current setting.
SDI_AIN_MODE_DIFF	This refers to Differential inputs.
SDI_AIN_MODE_SE	This refers to Single Ended inputs.
SDI_AIN_MODE_VREF	This refers to the Voltage Reference (VREF) input test mode.
SDI_AIN_MODE_ZERO	This refers to the zero-voltage input test mode.

4.7.6. SDI_IOCTL_AIN_RANGE

This service configures the board for a particular analog input voltage range.

Usage

Argument	Description
request	SDI_IOCTL_AIN_RANGE
arg	s32*

Valid argument options are as follows.

Value	Description
-1	This returns the current setting.
SDI_AIN_RANGE_1_25V	This refers to the ± 1.25 volt input range.
SDI_AIN_RANGE_2_5V	This refers to the ± 2.5 volt input range.
SDI_AIN_RANGE_5V	This refers to the ± 5 volt input range.
SDI_AIN_RANGE_10V	This refers to the ± 10 volt input range.

4.7.7. SDI_IOCTL_AUTOCAL

This service initiates an Autocalibration cycle. The service returns when the cycle completes.

NOTE: This service overwrites the current interrupt selection in order to detect the Autocalibration Done interrupt.

NOTE: When an error is encountered, the service writes a brief, descriptive error message to the system log.

Usage

Argument	Description
request	SDI_IOCTL_AUTOCAL
arg	Not used.

4.7.8. SDI_IOCTL_AUTOCAL_STATUS

Usage

Argument	Description
request	SDI_IOCTL_AUTOCAL_STATUS
arg	s32*

Valid argument options returned are as follows.

Value	Description
SDI_AUTOCAL_STATUS_ACTIVE	Autocalibration is in progress.
SDI_AUTOCAL_STATUS_FAIL	Autocalibration failed.
SDI_AUTOCAL_STATUS_PASS	Autocalibration passed.

4.7.9. SDI_IOCTL_CH_GRP_X_SRC

This service selects the clock source for the specified channel group. The channel group is specified by using one of the below listed IOCTL command codes. An error will be returned either for attempting to configure a channel group not support on the board, or for selecting a source not supported on the board. For clarification refer to the hardware reference manual for your board.

Command Code	Description
SDI_IOCTL_CH_GRP_0_SRC	This refers to Channel Group 0.
SDI_IOCTL_CH_GRP_1_SRC	This refers to Channel Group 1.
SDI_IOCTL_CH_GRP_2_SRC	This refers to Channel Group 2, which is not present on all boards.
SDI_IOCTL_CH_GRP_3_SRC	This refers to Channel Group 3, which is not present on all boards.

Usage

Argument	Description
request	SDI_IOCTL_CH_GRP_X_SRC
arg	s32*

Valid argument options are as follows. Some options are not available on all boards. For clarification refer to the hardware reference manual for your board.

Value	Description
-1	This returns the current setting.
SDI_CH_GRP_SRC_DISABLE	This disables the input from the group's analog inputs.
SDI_CH_GRP_SRC_EXTERN	This refers to the External Clock Input signal on the cable interface.
SDI_CH_GRP_SRC_GEN_A	This refers to Rate Generator A.
SDI_CH_GRP_SRC_GEN_B	This refers to Rate Generator B.
SDI_CH_GRP_SRC_GEN_C	This refers to Rate Generator C, which is not present on all boards.
SDI_CH_GRP_SRC_GEN_D	This refers to Rate Generator D, which is not present on all boards.

4.7.10. SDI_IOCTL_CHANNEL_ORDER

This service configures the board's channel number ordering option for converted data entering the Analog Input Buffer.

Usage

Argument	Description
request	SDI_IOCTL_CHANNEL_ORDER
arg	s32*

Valid argument options are as follows.

Value	Description
-1	This returns the current setting.
SDI_CHANNEL_ORDER_ASYNC	This refers asynchronous ordering in which data enters the Analog Input Buffer in an arbitrary order.
SDI_CHANNEL_ORDER_SYNC	This refers synchronous ordering in which data enters the Analog Input Buffer from the lowest active channel to the highest.

4.7.11. SDI_IOCTL_CHANNELS_READY

This service reports the state of the Channels Ready flag.

Usage

Argument	Description
request	SDI_IOCTL_CHANNELS_READY
arg	s32*

Valid argument options passed in are as follows.

Value	Description
-1	This returns the current setting.
SDI_CHANNELS_READY_WAIT	Wait up to one second for the flag to become set.

Valid argument options returned are as follows.

Value	Description
SDI_CHANNELS_READY_NO	The Channels Ready flag is not set.
SDI_CHANNELS_READY_YES	The Channels Ready flag is set.

4.7.12. SDI_IOCTL_DATA_FORMAT

This service configures the board's digital encoding format for converted analog input.

Usage

Argument	Description
request	SDI_IOCTL_DATA_FORMAT
arg	s32*

Valid argument options are as follows.

Value	Description
-1	This returns the current setting.
SDI_DATA_FORMAT_2S_COMP	This refers to the Two's Complement format.
SDI_DATA_FORMAT_OFF_BIN	This refers to the Offset Binary format.

4.7.13. SDI_IOCTL_INIT_MODE

This service configures the board for either Initiator mode operation or Target mode operation.

NOTE: If the initialization service returns an error status, an error message will be posted to the system log briefly describing the error condition.

Usage

Argument	Description
request	SDI_IOCTL_INIT_MODE
arg	s32*

Valid argument options are as follows.

Value	Description
-1	This returns the current setting.
SDI_INIT_MODE_INITIATOR	This refers to Initiator mode operation.
SDI_INIT_MODE_TARGET	This refers to Target mode operation.

4.7.14. SDI_IOCTL_INITIALIZE

This service returns all driver interface settings for the board to the state they were in when the board was first opened. This includes both hardware-based settings and software-based settings.

Usage

Argument	Description
request	SDI_IOCTL_INITIALIZE
arg	Not used.

4.7.15. SDI_IOCTL_IRQ_SEL

This service selects the interrupt option for the firmware interrupt.

Usage

Argument	Description
request	SDI_IOCTL_IRQ_SEL
arg	s32*

Valid argument options are as follows.

Value	Description
-1	This returns the current setting.
SDI_IRQ_AUTOCAL_DONE	This refers to Autocalibration completion.
SDI_IRQ_BUF_AE	This refers to the Almost Empty state of the buffer.
SDI_IRQ_BUF_AF	This refers to the Almost Full state of the buffer.
SDI_IRQ_BUF_THRESH_H2L	This refers to a high-to-low transition of the buffer threshold status.
SDI_IRQ_BUF_THRESH_L2H	This refers to a low-to-high transition of the buffer threshold status.
SDI_IRQ_CHANNELS_READY	This refers to assertion of the Channels Ready status.
SDI_IRQ_INIT_DONE	This refers to initialization completion.

4.7.16. SDI_IOCTL_QUERY

This service queries the driver for information about the device. The query is performed by passing in the identifier for the information desired, which is where the response to the query is placed upon completion.

Usage

Argument	Description
request	SDI_IOCTL_QUERY
arg	s32*

Valid argument values passed to the service are as follows. A return value of -1 indicates that the option passed to the service is unrecognized.

Value	Description
SDI_QUERY_AUTOCAL_MS	This refers to the maximum duration of an Autocalibration cycle, in milliseconds.
SDI_QUERY_CHANNEL_D16	This refers to the meaning of bit D16 in the Board Revision Register. A return value of zero indicates that bit D16 is reserved. A positive value indicates the number of channels that D16 refers to.
SDI_QUERY_CHANNEL_D17	This refers to the meaning of bit D17 in the Board Revision Register. A return value of zero indicates that bit D17 is reserved. A positive value indicates the number of channels that D17 refers to.
SDI_QUERY_CHANNEL_GPS	This refers to the number of Channel Groups present on the board.

SDI_QUERY_CHANNEL_MAX	This refers to the maximum number channels supported by this model board, though the number of channels present may be less.
SDI_QUERY_CHANNEL_QTY	This refers to the number of channels present on the board.
SDI_QUERY_COUNT	This refers to the number of supported query options.
SDI_QUERY_DEVICE_TYPE	This refers to the base model type for the board. See below for valid return values.
SDI_QUERY_DMDMA	This refers to the board's support for Demand Mode DMA. A value of zero indicates that it is not supported. A non-zero value indicates that it is supported.
SDI_QUERY_FGEN_MAX	This refers to the board's maximum supported FGEN frequency.
SDI_QUERY_FGEN_MIN	This refers to the board's minimum supported FGEN frequency.
SDI_QUERY_FIFO_SIZE	This refers to the size of the FIFO, which is the Analog Input Buffer.
SDI_QUERY_FORM_FACTOR	This refers to the board's form factor.
SDI_QUERY_FSAMP_MAX	This refers to the board's maximum supported sample rate, FSAMP.
SDI_QUERY_FSAMP_MIN	This refers to the board's minimum supported sample rate, FSAMP.
SDI_QUERY_INIT_MS	This refers to the maximum duration of an Initialization cycle, in milliseconds.
SDI_QUERY_IRQ_MASK	This refers to the bitmap of supported interrupt sources. †
SDI_QUERY_NDIV_MASK	This refers to the bitmap mask for the width of an NDIV field.
SDI_QUERY_NDIV_MAX	This refers to the board's maximum supported NDIV value.
SDI_QUERY_NDIV_MIN	This refers to the board's minimum supported NDIV value.
SDI_QUERY_NRATE_MASK	This refers to the bitmap mask for the width of an NRATE field.
SDI_QUERY_NRATE_MAX	This refers to the board's maximum supported NRATE value.
SDI_QUERY_NRATE_MIN	This refers to the board's minimum supported NRATE value.
SDI_QUERY_RATE_GEN_QTY	This refers to the number of Rate Generators present on the board.
SDI_QUERY_THRESHOLD_MASK	This refers to the bitmap mask for the width of the Buffer Threshold field.

† An interrupt source in the Interrupt Control Register is supported if the bit corresponding to the source index number is set in the mask. For example, option index three corresponds to bit D3 (code wise, bit = $1 \ll \text{index}$).

Valid device types are as follows and are independent of the form factor or ordered options.

Value	Description
GSC_DEV_TYPE_6SDI	This refers to 6SDI models regardless of the form factor.
GSC_DEV_TYPE_16SDI	This refers to 16SDI models regardless of the form factor.
GSC_DEV_TYPE_16SDI_HS	This refers to 16SDI-HS models regardless of the form factor. †
GSC_DEV_TYPE_16HSDI	This refers to 16HSDI models regardless of the form factor. †

† The PMC-16SDI-HS and the PMC-16HSDI are not distinguishable via software so are both reported to be 16HSDI type boards.

4.7.17. SDI_IOCTL_REG_MOD

This service performs a read-modify-write on the specified 16SDI register. This includes only the GSC specific registers. All PCI and PLX Feature Set Registers are read-only. Refer to `16sdi.h` for a complete list of the accessible registers.

Usage

Argument	Description
request	SDI_IOCTL_REG_MOD
arg	<code>gsc_reg_t*</code>

Definition

```
typedef struct
{
    u32 reg;      // range: any valid register definition
    u32 value;    // range: 0x0-0xFFFFFFFF
    u32 mask;     // range: 0x0-0xFFFFFFFF
} gsc_reg_t;
```

Fields	Description
reg	This is set to the identifier for the register to access.
value	This is the value to write to the modified register bits.
mask	This is a mask of bits to be modified. Bits which are set are those to be modified.

4.7.18. SDI_IOCTL_REG_READ

This service reads the value of a 16SDI register. This includes all PCI registers, all PLX Feature Set Registers, and all GSC specific registers. Refer to `16sdi.h` and `gsc_pci9080.h` for a complete list of the accessible registers. All registers are accessed by their native size.

Usage

Argument	Description
request	SDI_IOCTL_REG_READ
arg	<code>gsc_reg_t*</code>

Definition

```
typedef struct
{
    u32 reg;      // range: any valid register definition
    u32 value;    // range: 0x0-0xFFFFFFFF
    u32 mask;     // This is unused for register reads.
} gsc_reg_t;
```

Fields	Description
reg	This is set to the identifier for the register to access.
value	The value from the register is returned here.
mask	This field is ignored for register reads.

4.7.19. SDI_IOCTL_REG_WRITE

This service writes a value to a 16SDI register. This includes only the GSC specific registers. All PCI and PLX Feature Set Registers are read-only. Refer to `16sdi.h` for a complete list of the accessible registers.

Usage

Argument	Description
request	SDI_IOCTL_REG_WRITE
arg	<code>gsc_reg_t*</code>

Definition

```
typedef struct
{
```

```

u32 reg;    // range: any valid register definition
u32 value;  // range: 0x0-0xFFFFFFFF
u32 mask;   // This is unused for register writes.
} gsc_reg_t;

```

Fields	Description
reg	This is set to the identifier for the register to access.
value	This is the value to write to the register.
mask	This field is ignored for register writes.

4.7.20. SDI_IOCTL_RX_IO_ABORT

This service aborts an ongoing `sdi_read()` request.

Usage

Argument	Description
request	SDI_IOCTL_RX_IO_ABORT
arg	s32*

The results are reported as one of the following values.

Value	Description
SDI_IO_ABORT_NO	A read request was not aborted as none were ongoing.
SDI_IO_ABORT_YES	An ongoing read request was aborted.

4.7.21. SDI_IOCTL_RX_IO_MODE

This service sets the data transfer mode used by the `sdi_read()` service to retrieve data from the board.

Usage

Argument	Description
request	SDI_IOCTL_RX_IO_MODE
arg	s32*

Valid argument values are as follows.

Value	Description
-1	This option returns the current setting.
GSC_IO_MODE_BMDMA	This refers to Block Mode DMA. With this mode a DMA request is initiated either when the Buffer Threshold is met or when the Analog Input Buffer contains enough data to complete the request, whichever occurs first. This is the default.
GSC_IO_MODE_DMDMA	This refers to Demand Mode DMA. In this case the DMA is started immediately though the data is actually transferred as it becomes available in the Analog Input Buffer. This is generally the most efficient method.
GSC_IO_MODE_PIO	This refers to Programmed I/O, which uses repetitive register accesses. With this mode data is transferred as it becomes available in the Analog Input Buffer. This is generally the least efficient method.

4.7.22. SDI_IOCTL_RX_IO_TIMEOUT

This service sets the timeout period for read requests in seconds. The default is 10 seconds.

Usage

Argument	Description
request	SDI_IOCTL_RX_IO_TIMEOUT
arg	s32*

Valid argument values are in the range from zero to 3600, -1, and SDI_IOCTL_TIMEOUT_INFINITE. A value of zero tells the driver not to sleep in order to wait for more data, and should only be used with PIO mode reads. A value of -1 is used to retrieve the current setting. If the option SDI_IOCTL_TIMEOUT_INFINITE is used, then the driver will wait indefinitely rather than timing out. The default is 10 seconds.

4.7.23. SDI_IOCTL_RATE_DIV_XX_NDIV

This service sets the NDIV value for the specified Rate Divider. The Rate Divider is specified by using one of the below listed IOCTL command codes. An error will be returned either for attempting to configure a Rate Divider not support on the board, or for attempting to configure a Rate Divider with a value not supported on the board. For clarification refer to the hardware reference manual for your board.

Command Code	Description
SDI_IOCTL_RATE_DIV_00_NDIV	This refers to Rate Divider 0.
SDI_IOCTL_RATE_DIV_01_NDIV	This refers to Rate Divider 1.
SDI_IOCTL_RATE_DIV_02_NDIV	This refers to Rate Divider 2.
SDI_IOCTL_RATE_DIV_03_NDIV	This refers to Rate Divider 3.
SDI_IOCTL_RATE_DIV_04_NDIV	This refers to Rate Divider 4.
SDI_IOCTL_RATE_DIV_05_NDIV	This refers to Rate Divider 5.
SDI_IOCTL_RATE_DIV_06_NDIV	This refers to Rate Divider 6.
SDI_IOCTL_RATE_DIV_07_NDIV	This refers to Rate Divider 7.
SDI_IOCTL_RATE_DIV_08_NDIV	This refers to Rate Divider 8.
SDI_IOCTL_RATE_DIV_09_NDIV	This refers to Rate Divider 9.
SDI_IOCTL_RATE_DIV_10_NDIV	This refers to Rate Divider 10.
SDI_IOCTL_RATE_DIV_11_NDIV	This refers to Rate Divider 11.
SDI_IOCTL_RATE_DIV_12_NDIV	This refers to Rate Divider 12.
SDI_IOCTL_RATE_DIV_13_NDIV	This refers to Rate Divider 13.
SDI_IOCTL_RATE_DIV_14_NDIV	This refers to Rate Divider 14.
SDI_IOCTL_RATE_DIV_15_NDIV	This refers to Rate Divider 15.

Usage

Argument	Description
request	SDI_IOCTL_RATE_DIV_XX_NDIV
arg	s32*

Valid argument values are from the minimum NDIV supported to the maximum NDIV supported, as well as -1 which retrieves the current setting.

4.7.24. SDI_IOCTL_RATE_GEN_X_NRATE

This service sets the NRATE value for the specified Rate Generator. The Rate Generator is specified by using one of the below listed IOCTL command codes. An error will be returned either for attempting to configure a Rate

Generator not support on the board, or for attempting to configure a Rate Generator with a value not supported on the board. For clarification refer to the hardware reference manual for your board.

Command Code	Description
SDI_IOCTL_RATE_GEN_A_NRATE	This refers to Rate Generator A.
SDI_IOCTL_RATE_GEN_B_NRATE	This refers to Rate Generator B.
SDI_IOCTL_RATE_GEN_C_NRATE	This refers to Rate Generator C, which is not present on all boards.
SDI_IOCTL_RATE_GEN_D_NRATE	This refers to Rate Generator D, which is not present on all boards.

Usage

Argument	Description
request	SDI_IOCTL_RATE_GEN_X_NRATE
arg	s32*

Valid argument values are from the minimum NRATE supported to the maximum NRATE supported, as well as -1 which retrieves the current setting.

4.7.25. SDI_IOCTL_SW_SYNC

This service initiates a synchronization pulse on the cable's Sync Output signal. The pulse is from 60ns to 200ns wide.

Usage

Argument	Description
request	SDI_IOCTL_SW_SYNC
arg	Not used.

4.7.26. SDI_IOCTL_SW_SYNC_MODE

This service configures the action the board takes upon either generating or receiving a synchronization pulse.

Usage

Argument	Description
request	SDI_IOCTL_SW_SYNC_MODE
arg	s32*

Valid argument values are as follows.

Value	Description
-1	This option returns the current setting.
SDI_SW_SYNC_MODE_CLR_BUF	This refers to clearing the Analog Input Buffer.
SDI_SW_SYNC_MODE_SW_SYNC	This refers to synchronizing ADC operation to the current clock.

4.7.27. SDI_IOCTL_WAIT_CANCEL

This service resumes all threads blocked via SDI_IOCTL_WAIT_EVENT IOCTL calls (section 4.7.28, page 34), according to the provided criteria. When a blocked thread is waiting for any event specified in the structure, then the thread is resumed.

NOTE: The driver itself makes use of the wait services for various internal operations. Driver initiated waits are unaffected by application cancel requests.

Usage

Argument	Description
request	SDI_IOCTL_WAIT_CANCEL
arg	gsc_wait_t*

Definition

```
typedef struct
{
    u32  flags;
    u32  main;
    u32  gsc;
    u32  alt;
    u32  io;
    u32  timeout_ms;
    u32  count;
} gsc_wait_t;
```

Fields	Description
flags	This is unused by wait cancel operations.
main	This specifies the set of GSC_WAIT_MAIN_* events whose wait requests are to be cancelled. Refer to section 4.7.28.2 on page 35.
gsc	This specifies the set of SDI_WAIT_GSC_* events whose wait requests are to be cancelled. Refer to section 4.7.28.3 on page 36.
alt	This is unused by the 16SDI driver and should be zero.
io	This specifies the set of GSC_WAIT_IO_* events whose wait requests are to be cancelled. Refer to section 4.7.28.4 on page 36.
timeout_ms	This is unused by wait cancel operations.
count	Upon return this indicates the number of waits that were cancelled.

4.7.28. SDI_IOCTL_WAIT_EVENT

This service blocks a thread until any one of a specified set of events occurs, or until a timeout lapses, whichever occurs first. The set of possible events to wait for are specified in the structure's `main`, `gsc`, `alt` and `io` fields. All field values must be valid and at least one event must be specified. If the thread is resumed because one of the referenced events has occurred, then the bit for the respective event is the only event bit that will be set. All other event bits and fields will be zero. (Multiple event bits will be set only if the events occur simultaneously.)

NOTE: The service waits only for the first of the specified events, not for all specified events.

NOTE: A wait timeout is reported via the `gsc_wait_t` structure's `flags` field having the `GSC_WAIT_FLAG_TIMEOUT` flag set, rather than via an `ETIMEDOUT` error.

Usage

Argument	Description
request	SDI_IOCTL_WAIT_EVENT
arg	gsc_wait_t*

Definition

```
typedef struct
{
    u32  flags;
    u32  main;
    u32  gsc;
    u32  alt;
    u32  io;
    u32  timeout_ms;
    u32  count;
} gsc_wait_t;
```

Fields	Description
flags	This must initially be zero. Upon return this indicates the reason that the thread was resumed. Refer to section 4.7.28.1 on page 35.
main	This specifies any number of GSC_WAIT_MAIN_* events that the thread is to wait for. Refer to section 4.7.28.2 on page 35.
gsc	This specifies any number of SDI_WAIT_GSC_* events that the thread is to wait for. Refer to section 4.7.28.3 on page 36.
alt	This is unused by the 16SDI driver and must be zero.
io	This specifies any number of GSC_WAIT_IO_* events that the thread is to wait for. Refer to section 4.7.28.4 on page 36.
timeout_ms	This specified the maximum amount of time, in milliseconds, that the thread is to wait for any of the referenced events. A value of zero means do not timeout at all. If non-zero, then upon return the value will be the approximate amount of time actually waited.
count	This is unused by wait event operations and must be zero.

4.7.28.1. gsc_wait_t.flags Options

Upon return from a wait request the wait structure's flags field will indicate the reason that the thread was resumed. Only one of the below options will be set.

Fields	Description
GSC_WAIT_FLAG_CANCEL	The wait request was cancelled.
GSC_WAIT_FLAG_DONE	One of the referenced events occurred.
GSC_WAIT_FLAG_TIMEOUT	The timeout period lapsed before a referenced event occurred.

4.7.28.2. gsc_wait_t.main Options

The wait structure's main field may specify any of the below primary interrupt options. These interrupt options are supported by the 16SDI and other General Standards products.

Fields	Description
GSC_WAIT_MAIN_DMA0	This refers to the DMA Done interrupt on DMA engine number zero.
GSC_WAIT_MAIN_DMA1	This refers to the DMA Done interrupt on DMA engine number one.
GSC_WAIT_MAIN_GSC	This refers to any of the Interrupt Control/Status Register interrupts.
GSC_WAIT_MAIN_OTHER	This generally refers to an interrupt generated by another device sharing the same interrupt as the 16SDI.
GSC_WAIT_MAIN_PCI	This refers to any interrupt generated by the 16SDI.
GSC_WAIT_MAIN_SPURIOUS	This refers to board interrupts which should never be generated.
GSC_WAIT_MAIN_UNKNOWN	This refers to board interrupts whose source could not be identified.

4.7.28.3. `gsc_wait_t.gsc` Options

The wait structure's `gsc` field may specify any combination of the below interrupt options. These are the interrupt options referenced in the Board Control Register. Applications are responsible for selecting the desired interrupt options. Refer to `SDI_IOCTL_IRQ_SEL` (section 4.7.15, page 28).

Value	Description
<code>SDI_WAIT_GSC_AUTOCAL_DONE</code>	This refers to Autocalibration completion.
<code>SDI_WAIT_GSC_BUF_AE</code>	This refers to the Almost Empty state of the buffer.
<code>SDI_WAIT_GSC_BUF_AF</code>	This refers to the Almost Full state of the buffer.
<code>SDI_WAIT_GSC_BUF_THRESH_H2L</code>	This refers to a high-to-low transition of the buffer threshold status.
<code>SDI_WAIT_GSC_BUF_THRESH_L2H</code>	This refers to a low-to-high transition of the buffer threshold status.
<code>SDI_WAIT_GSC_CHANNELS_READY</code>	This refers to assertion of the Channels Ready status.
<code>SDI_WAIT_GSC_INIT_DONE</code>	This refers to initialization completion.

4.7.28.4. `gsc_wait_t.io` Options

The wait structure's `io` field may specify any of the below event options. These events are generated in response to application board data read requests.

Fields	Description
<code>SDI_WAIT_IO_RX_ABORT</code>	This refers to read requests which have been aborted.
<code>SDI_WAIT_IO_RX_DONE</code>	This refers to read requests which have been satisfied.
<code>SDI_WAIT_IO_RX_ERROR</code>	This refers to read requests which end due to an error.
<code>SDI_WAIT_IO_RX_TIMEOUT</code>	This refers to read requests which end due to the timeout period lapse.

4.7.29. `SDI_IOCTL_WAIT_STATUS`

This service counts all threads blocked via the `SDI_IOCTL_WAIT_EVENT` IOCTL service (section 4.7.28, page 34), according to the provided criteria. A match is made when a waiting thread's wait criteria matches any of the criteria specified in the structure passed to this service.

NOTE: The driver itself makes use of the wait services for various internal operations. Driver initiated waits are ignored by application status requests.

Usage

Argument	Description
<code>request</code>	<code>SDI_IOCTL_WAIT_STATUS</code>
<code>arg</code>	<code>gsc_wait_t*</code>

Definition

```
typedef struct
{
    u32  flags;
    u32  main;
    u32  gsc;
    u32  alt;
    u32  io;
    u32  timeout_ms;
    u32  count;
} gsc_wait_t;
```

Fields	Description
flags	This is unused by wait status operations.
main	This specifies the set of GSC_WAIT_MAIN_* events whose wait requests are to be counted. Refer to section 4.7.28.2 on page 35.
gsc	This specifies the set of SDI_WAIT_GSC_* events whose wait requests are to be counted. Refer to section 4.7.28.3 on page 36.
alt	This is unused by the 16SDI driver and should be zero.
io	This specifies the set of GSC_WAIT_IO_* events whose wait requests are to be counted. Refer to section 4.7.28.4 on page 36.
timeout_ms	This is unused by wait status operations.
count	Upon return this indicates the number of waits that met any of the specified criteria.

5. The Driver

NOTE: Contact General Standards Corporation if additional driver functionality is required.

5.1. Files

The device driver files are summarized in the table below.

Description	Files	Location
Source Files	*.c, *.h ...	.../driver/
Header File	16sdi.h	
Driver File	16sdi.ko † 16sdi.o ‡	

† This is for kernel versions 2.6 and later.

‡ This is for kernel versions 2.4 are earlier.

5.2. Build

NOTE: Building the driver requires installation of the kernel headers and possibly other packages.

The device driver is built via the Overall Make Script (section 2.7, page 12), but can be built separately following the below steps.

1. Change to the directory where the driver and its sources are installed (.../driver/).
2. Remove existing build targets by issuing the below command.

```
make clean
```

3. Build the driver by issuing the below command.

```
make
```

NOTE: Due to the differences between the many Linux distributions some build errors may occur. These errors may include system header location differences, which should be easily corrected.

5.3. Startup

NOTE: The driver will have to be built before being used as it is provided in source form only.

The startup script used in this procedure is designed to load the device driver and create fresh device nodes. This is accomplished by unloading the current driver, if loaded, and then loading the accompanying driver executable. In addition, the script deletes and recreates the device nodes. This is done to ensure that the device nodes in use have the same major number as assigned dynamically to the driver by the kernel, and so that the number of device nodes corresponds to the number of boards identified by the driver.

5.3.1. Manual Driver Startup Procedures

Start the driver manually by following the below listed steps.

NOTE: The following steps may require elevated privileges.

1. Change to the directory where the driver sources are installed (.../driver/).
2. Install the driver module and create the device nodes by executing the below command. If any errors are encountered then an appropriate error message will be displayed.

```
./start
```

NOTE: This script must be executed each time the host is booted.

NOTE: The 16SDI device node major number is assigned dynamically by the kernel. The minor numbers and the device node suffix numbers are index numbers beginning with zero, and increase by one for each additional board installed.

3. Verify that the device driver module has been loaded by issuing the below command and examining the output. The module name `16sdi` should be included in the output.

```
lsmod
```

4. Verify that the device nodes have been created by issuing the below command and examining the output. The output should include one node for each installed board.

```
ls -l /dev/16sdi.*
```

5.3.2. Automatic Driver Startup Procedures

Start the driver automatically with each system reboot by following the below listed steps.

1. Locate and edit the system startup script `rc.local`, which should be in the `/etc/rc.d/` directory. Modify the file by adding the below line so that it is executed with every reboot. The example is based on the driver being installed in `/usr/src/linux/drivers/`, though it may have been installed elsewhere.

```
/usr/src/linux/drivers/16sdi/driver/start
```

NOTE: For `systemd` installations the file `rc.local` may be located under the `/etc/` directory rather than under `/etc/rc.d/`.

2. Load the driver and create the required device nodes by rebooting the system.
3. Verify that the driver is loaded and that the device nodes have been created. Do this by following the verification steps given in the manual startup procedures.

5.3.2.1. File `rc.local` Not Present

Some distributions may not install a default version of `rc.local`. Some may not even create the directory `/etc/rc.d/`. If the directory is not present, then it may be created. The directory must be created with the owner and group set to `root`. The directory permissions must be set to `rxwxr-xr-x`. If the file `/etc/rc.d/rc.local` is not present, then it too may be created. The file must also be created with the owner and group set to `root`. Additionally, the file permissions must also be set to `rxwxr-xr-x`. After the directory and file are created as described, reboot to verify boot time loading of the driver. Here is an example of a default version of `rc.local`.

```
#!/bin/bash

# Add your local content here.
```

5.3.2.2. Default `rc.local` File Permissions

The `rc.local` script may fail to run at boot time because some distributions install a default version of the file without execute permissions. Without execute permissions, boot time invocation of the script fails, which inhibits boot time loading of the driver. If this is the case, then change the file permissions to `rxwxr-xr-x`. After the file permissions are adjusted as described, reboot to verify boot time loading of the driver.

5.3.2.3. `systemd` Installations

With the advent of the `systemd` startup implementation, `rc.local` may be accessed via a `systemd` startup service. The service name may be `rc-local`, `rc-local.service` or something similar. This service may or may not be enabled by default. If the service is disabled, then the script will not execute, which prevents boot time loading of the driver. The service can be enabled with the below command line. After the service is enabled, reboot to verify boot time loading of the driver.

```
systemctl enable rc-local
```

NOTE: For `systemd` installations the file `rc.local` may be located under the `/etc/` directory rather than under `/etc/rc.d/`.

5.3.2.4. `systemd` and `rc.local` Timing

If the above steps have been performed but the driver still does not start then examine the `dmesg` output for driver messages. If the output shows that the driver starts and immediately stops, then the problem may be timing. That is, since `systemd` doesn't serialize startup initialization as done in the past, driver loading may fail if required services have not completed their own initialization. If this is the problem, then it may be corrected simply by inserting a delay in `rc.local` prior to it calling the driver's start script (i.e., sleep for one or more seconds).

5.3.2.5. SELinux Implications

If not disabled, then SELinux may prevent boot time loading of the driver. If this is the case, then it can be verified and corrected using SELinux related tools and utilities. First, install the necessary software using the below command. (As necessary, replace the `yum` command line with that which is available for your distribution.)

```
yum install setroubleshoot setools
```

Next, run the below command to determine if SELinux is preventing the driver from loading at boot time.

```
sealert -a /var/log/audit/audit.log
```

If SELinux is preventing the driver from loading, then the output from the above command should include a reference to the driver's start script, the `insmod` command that loads the driver or the name of the driver executable. If so, then the output should also indicate the commands necessary to resolve the issue. The following is an example of the instructions given when the culprit is `insmod`, which is the start script command that loads the driver. After running these commands reboot the system to verify boot time loading of the driver.

```
ausearch -c 'insmod' --raw | audit2allow -M my-insmod
semodule -X 300 -i my-insmod.pp
```

5.4. Verification

Follow the below steps to verify that the driver has been properly installed and started.

1. Verify that the file `/proc/16sdi` is present. If the file is present then the driver is loaded and running. Verify the file's presence by viewing its content with the below command.

```
cat /proc/16sdi
```

5.5. Version

The driver version number can be obtained in a variety of ways. It is reported by the driver both when the driver is loaded and when it is unloaded (depending on kernel configuration options, this may be visible only in places such as `/var/log/messages`). It is reported in the text file `/proc/16sdi` while the driver is loaded and running. The version number is also given in the file `release.txt` in the root install directory.

5.6. Shutdown

Shutdown the driver following the below listed steps.

NOTE: The following steps may require elevated privileges.

1. If the driver is currently loaded then issue the below command to unload the driver.

```
rmmod 16sdi
```

2. Verify that the driver module has been unloaded by issuing the below command. The module name `16sdi` should not be in the listed output.

```
lsmod
```

6. Document Source Code Examples

The source code examples included in this document are built into a statically linkable library usable with console applications. The purpose of these files is to verify that the documentation samples compile and to provide a library of working sample code to assist in a user's learning curve and application development effort.

6.1. Files

The library files are summarized in the table below.

Description	Files	Location
Source Files	*.c, *.h ...	.../docsrc/
Header File	16sdi_dsl.h	.../include/
Library File	16sdi_dsl.a	.../lib/

6.2. Build

The library is built via the Overall Make Script (section 2.7, page 12), but can be built separately following the below steps.

1. Change to the directory where the documentation sources are installed (.../docsrc/).
2. Remove existing build targets by issuing the below command.

```
make clean
```

3. Compile the sample files and build the library by issuing the below command.

```
make
```

4. Rebuild the Main Library (section 3.2.1, page 15).

6.3. Library Use

The library is used both at application compile time and at application link time. At compile time include the above listed header file in each source file using a component of the library interface. At link time include the above listed static library file with the objects being linked with the application.

7. Utilities Source Code

The API Library installation includes a body of utility source code designed to aid in the understanding and use of the interface calls and IOCTL services. Utility sources are also included for device independent and common, general-purpose services. Most of the utilities are implemented as visual wrappers around the corresponding services to facilitate structured console output for the sample applications. The utility sources are compiled and linked into static libraries to simplify their use. An additional purpose of these utility services is to provide a library of working sample code to assist in a user's learning curve and application development effort.

For each API function there is a corresponding utility source file with a corresponding utility service. As an example, for the API function `sdi_open()` there is the utility file `open.c` containing the utility function `sdi_open_util()`. The naming pattern is as follows: API function `sdi_xxxx()`, utility file name `xxxx.c`, utility function `sdi_xxxx_util()`. Additionally, for each IOCTL code there is a corresponding utility source file with a corresponding utility service. As an example, for IOCTL code `SDI_IOCTL_QUERY` there is the utility file `query.c` containing the utility function `sdi_query()`. The naming pattern is as follows: IOCTL code `SDI_IOCTL_XXXX`, utility file name `xxxx.c`, utility function `sdi_xxxx()`.

7.1. Files

The utility files are summarized in the table below.

Description	Files	Location
Source Files	*.c, *.h ...	.../utils/
Header File	16sdi_utils.h	.../include/
Library Files	16sdi_utils.a gsc_utils.a os_utils.a plx_utils.a	.../lib/

7.2. Build

The libraries are built via the Overall Make Script (section 2.7, page 12), but can be built separately following the below steps.

1. Change to the directory where the utility sources are installed (.../utils/).
2. Remove existing build targets by issuing the below command.

```
make clean
```

3. Compile the sample files and build the library by issuing the below command.

```
make
```

4. Rebuild the Main Library (section 3.2.1, page 15).

7.3. Library Use

The library is used both at application compile time and at application link time. At compile time include the above listed header file in each source file using a component of the library interface. At link time include the above listed static library file with the objects being linked with the application.

8. Operating Information

This section explains some basic operational procedures for using the 16SDI. This is in no way intended to be a comprehensive guide. This is simply to address a very few issues relating to their use.

8.1. Debugging Aids

The driver package includes the following items useful for development and/or debugging aids.

8.1.1. Device Identification

When communicating with technical support complete device identification is virtually always necessary. The *id* example application is provided for this specific purpose. This is a text only console application. The output can be piped to a file, which can then be emailed to GSC technical support when requested. Locate the application as follows.

Description	File	Location
Application	<i>id</i>	.../id/

8.1.2. Detailed Register Dump

Among the utility services provided is a function to generate a detailed listing of device registers to the console. When used, the function is typically used to verify device configuration. In these cases, the function should be called after complete device configuration and before the first I/O call. When intended for sending to GSC tech support, please set the *detail* arguments to 1. The function arguments are as follows. The utility location is given in the subsequent table.

Argument	Description
<i>fd</i>	This is the file descriptor used to access the device.
<i>detail</i>	If non-zero the register dump will include details of each register field.

Description	File/Name	Location
Function	<i>sdi_reg_list()</i>	Source File
Source File	<i>reg.c</i>	.../utils/
Header File	<i>16sdi_utils.h</i>	.../include/
Library File	<i>16sdi_utils.a</i>	.../lib/

8.2. Analog Input Configuration

The basic steps for Analog Input configuration are illustrated in the utility function noted below. The table also gives the location of the source file, the header file and the corresponding library containing the executable code. The referenced files are included via the Main Header and Main Library.

Item	Name/File	Location
Function	<i>sdi_config_ai()</i>	Source File
Source File	<i>config_ai.c</i>	.../utils/
Header File	<i>16sdi_utils.h</i>	.../include/
Library File	<i>16sdi_utils.a</i>	.../lib/

8.3. Data Transfer Modes

All device I/O requests move data through intermediate driver buffers on its way between the device and application memory. The data is processed in chunks no larger than the size of this intermediate buffer. The process used to

perform this transfer is according to the I/O mode selection. Movement of data between the application buffers and the intermediate driver buffers is performed by the kernel.

8.3.1. PIO - Programmed I/O

In PIO mode the driver reads data by repetitive registers reads from the input data buffer register until either the request is satisfied or the I/O timeout expires, whichever occurs first.

8.3.2. BMDMA - Block Mode DMA

For Block Mode DMA the driver initiates DMA transfers only after a sufficient volume of data has been received into the input buffer. In this mode the volume is sufficient when it meets or exceeds the threshold value. After that amount of data is in the input buffer the driver initiates a DMA then sleeps until the DMA Done interrupt is received. Using this DMA mode, a user request typically consists of numerous individual DMA transfers.

8.3.3. DMDMA - Demand Mode DMA

This DMA mode is similar to the Block Mode, except that the transfer is initiated immediately. Here however, the actual movement of data occurs as the data becomes available in the buffer instead of after it has been accumulated. The size of each individual transfer though is limited to the Threshold value plus one. Using this DMA mode, a user request typically consists of a single individual DMA transfer.

8.4. Multi-Board Synchronization

Multi-board synchronization is a feature of the 16SDI that enables two or more boards to sample analog input data in lock-step. Exercising this feature requires the boards to operate synchronously from the same clock source. This is done using the clock and sync signals on the cable interface. Though there are numerous varying ways of configuring the boards and of wiring the signals, the two basic configurations are described below.

8.4.1. Star Configuration

The *star* configuration generally permits all boards in the setup to operate with the least possible phase shift from one board to the next. This is accomplished by configuring all the boards in an identical manner and by wiring the clock and sync signals so that they follow as identical a path as possible from the initiator's output to the input of the initiator and the targets. If there are four or more boards in the setup, an initiator and three or more targets, then the clock and sync signals must go directly from the initiator's output to a Clock Driver board, as illustrated in Figure 2. If there are only two or three boards in the setup, an initiator and one or two targets, then a Clock Driver board is not needed, as illustrated in Figure 3. The table below shows the board programming that is specific to the *star* configuration.

Setting	Initiator	Target(s)
Initiator Mode	Initiator	Target
Channel Order	Synchronous	Synchronous
Rate Group 0 Clock Source	External	External

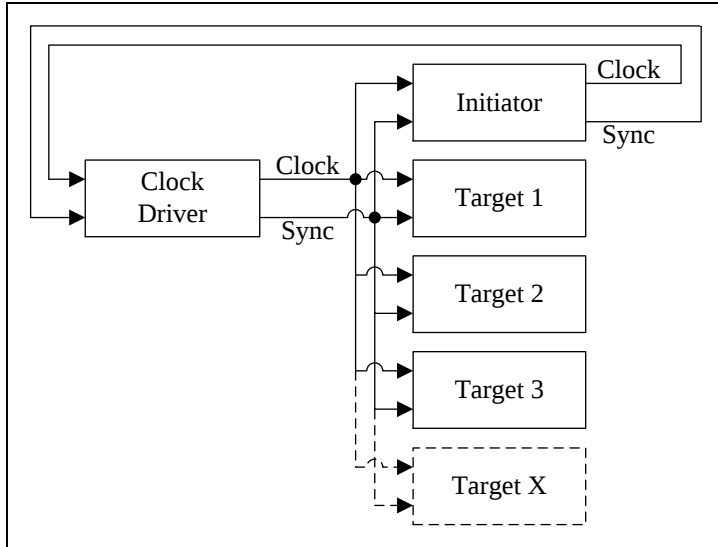


Figure 2 The *star* configuration with four or more boards requires a Clock Driver board.

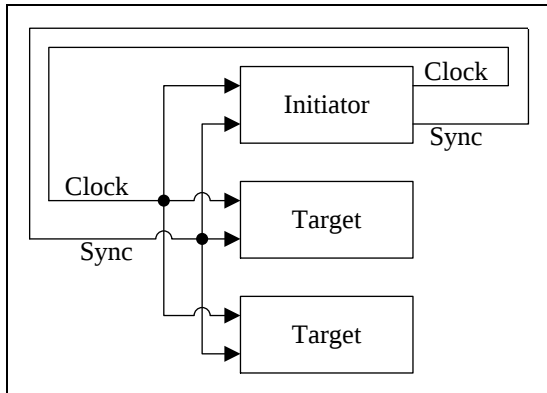


Figure 3 The *star* configuration with only two or three boards does not require a Clock Driver board.

8.4.2. Daisy Chain Configuration

The *daisy chain* configuration generally permits the most flexible placement of boards and wiring, and does not require a Clock Driver board. This is accomplished by configuring the boards and the wiring so that the clock and sync signals go from the initiator to the first target, then sequentially from the first target to the second and so on. This setup is applicable for any number of boards, as illustrated in Figure 4. The table below shows the board programming that is specific to the *daisy chain* configuration.

Setting	Initiator	Target(s)
Initiator Mode	Initiator	Target
Channel Order	Synchronous	Synchronous
Rate Group 0 Clock Source	Rate Generator A	External

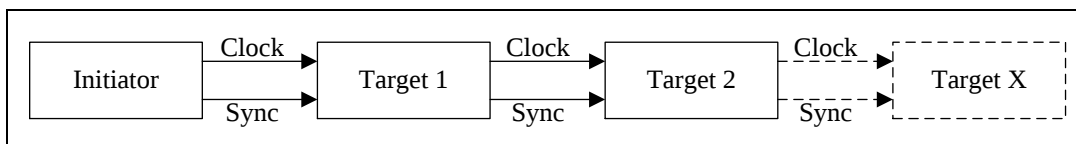


Figure 4 In this configuration the clock and sync signals are daisy chained from one board to the next.

9. Sample Applications

The driver archive includes a variety of sample and test applications located under the `samples` subdirectory. While they are provided without support and without any external documentation, any problems reported will be addressed as time permits. The applications are command line based and produce text output for display on a console. All of the applications are built via the Overall Make Script (section 2.7, page 12), but each may be built individually by changing to its respective directory and issuing the commands “`make clean`” and “`make`”. The initial output from each application includes information on its supported command line arguments. The following gives a brief overview of each application.

9.1. billion - Billion Byte Read - `.../billion/`

This application configures the designated board then reads in a billion bytes. The data is discarded after it is read.

9.2. clock - Clock Output - `.../clock/`

This application configures the board to drive the output clock signal for a user specified period of time. This is done to facilitate setup of test equipment to capture those signals during actual use.

9.3. fsamp - Sample Rate - `.../fsamp/`

This application reports the device configuration required to produce a user specified sample rate.

9.4. id - Identify Board - `.../id/`

This application reports detailed board identification information. This can be used with tech support to help identify as much technical information about the board as possible from software.

9.5. regs - Register Access - `.../regs/`

This application provides menu based interactive access to the board’s registers, and reports other pertinent information to the console.

9.6. rxrate - Receive Rate - `.../rxrate/`

This application configures the board for its highest ADC sample rate then reads the input as fast as possible. The purpose is to measure the peak sustainable input rate for the host, per the provided command line arguments.

9.7. savedata - Save Acquired Data - `.../savedata/`

This application configures the board for a modest sample rate, reads a megabyte of data, then saves the data to a hex file.

9.8. sbtest - Single Board Test - `.../sbtest/`

This application performs functional testing of the driver and a user specified board, at least to the extent possible with just a single board and no additional equipment.

9.9. sw_sync - Software Sync - `.../sw_sync/`

This application repetitively activates the board’s Software Sync feature and drives the output clock signal for a user specified period of time. This is done to facilitate setup of test equipment to capture those signals during actual use.

Document History

Revision	Description
October 21, 2024	Updated to version 5.7.111.50.0. Updated the kernel support table. Updated the description of the Autocalibration service. Minor editorial changes. Renamed SDI_GSC_ACVR to SDI_GSC_AVR. Removed the “util_” prefix from all generic utility sources. All Auto_Calibrate and similar content has been renamed to Autocal. All Auto_Cal_Status and similar content has been renamed to Autocal_Status.
October 25, 2022	Updated to version 5.6.101.44.0. Updated the kernel support table. Added section on environment variables. Updated the information for the open and close calls. Numerous minor editorial changes.
February 15, 2022	Updated to version 5.5.96.38.0. Updated the kernel support table. Minor editorial changes. Expanded automatic startup information.
September 11, 2019	Updated to version 5.4.87.28.0. Updated the kernel support table. Minor editorial changes. Added a licensing subsection. Added WAIT_EVENT note.
March 1, 2019	Updated to version 5.3.82.26.0. Updated the inside cover page. Updated the CPU and kernel support section. Minor editorial changes. Updated Block Mode DMA macro and associated information. Updated the I/O wait options.
January 23, 2018	Updated to version 5.2.73.20.0. Implemented API Library. General reorganization of document.
December 1, 2016	Updated to version 5.1.68.18.0. Removed the <code>built</code> field from the <code>/proc</code> file. Updated the kernel support table. Updated the command line arguments for the <code>fsamp</code> , <code>rxrate</code> and <code>savedata</code> sample applications. Organized sample applications alphabetically. Organized sample applications alphabetically. Updated the usage of the Wait Event <code>timeout_ms</code> field. Updated material on the open call. Added open access mode descriptions. Added support for infinite I/O timeouts. Updated the operating information section. Made various miscellaneous updates. Some document reorganization.
September 15, 2015	Updated to version 5.0.60.8.0. Updated the device node name to include a period before the device index. Removed double underscore that prefaced various data types. Added the Autocalibration Status IOCTL service. Added the Channels Ready IOCTL service. Added the Buffer Threshold Status IOCTL service.
February 28, 2014	Updated to version 4.5.52.0. Updated the kernel support table.
January 9, 2014	Updated to version 4.4.51.0. Updated the kernel support table.
November 14, 2013	Updated to version 4.4.50.0. Removed the <code>test</code> sample application.
July 16, 2013	Updated to version 4.4.45.0. Updated the kernel support table.
July 19, 2012	Updated to version 4.4.39.0. Updated the kernel support table.
December 20, 2011	Updated to version 4.3.34.0.
November 2, 2011	Updated to version 4.2.32.0. Various editorial changes. Added the IRQ_SEL IOCTL service. Updated the CPU and Kernel Support information. Updated the comments for the Initialize IOCTL service. Changed the spelling of various Autocalibration related software items. Added Form Factor and IRQ Mask query options.
December 28, 2009	Updated to version 4.1.13.0.
April 17, 2009	Updated to version 4.0.5.0.
April 17, 2009	Updated to version 4.0.4.0.
November 11, 2008	Updated to version 4.0.0.0. This was a major driver update with significant interface modifications. A number of sample applications were also added.
May 12, 2006	Added IOCTL_SDI_SYNCHRONIZE_SCAN.
October 12, 2005	Added IOCTL_SDI_FILL_BUFFER.
September 20, 2004	Added references to Linux kernel version 2.6.
February 25, 2003	Typos corrected.
January 31, 2003	Initial release.