16AO20

16-bit, 20/12/6 channel, 440K S/S/Ch Analog Output

PC104P-16AO20

Linux Device Driver User Manual

Manual Revision: May 19, 2009
Driver Release 2.0.7.1

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

Table of Contents

1. Introduction

This user manual applies to driver release 2.0.7.1.

1.1. Purpose

The purpose of this document is to describe the interface to the 16AO20 Linux device driver. This driver software provides the interface between "Application Software" and a 16AO20 board. The interface to the board is at the device level.

1.2. Acronyms

The following is a list of commonly occurring acronyms used throughout this document.

Acronyms	**Description**
DMA	Direct Memory Access
GSC	General Standards Corporation
PC104+	This refers to the PC104+ form factor of PCI boards.
PCI	Peripheral Component Interconnect
PMC	PCI Mezzanine Card

1.3. Definitions

The following is a list of commonly occurring terms used throughout this document.

Term	**Definition**
16AO20	This is used as a general reference to any board supported by this driver.
Application	This refers to user mode processes, which runs in user space with user mode privileges.
Driver	This refers to the kernel mode device driver, which runs in kernel space with kernel mode privileges.

1.4. Software Overview

The 16AO20 driver software executes under control of the Linux operating system and runs in Kernel Mode as a Kernel Mode device driver. The 16AO20 device driver is implemented as a standard dynamically loadable Linux device driver written in the C programming language. With the driver, user applications are able to open and close a device and, while open, perform write and I/O control operations. Read operations to the board are not supported.

1.5. Hardware Overview

The 16AO20 is a high-performance, 16-bit analog output board that incorporates 20, 12 or 6 output channels. The host side connection is PCI based and the form factor is according to the model ordered. The board is capable of outputting data at up to 440K samples per second over each channel. Internal clocking permits sampling rates from 440K samples per second down to 244 samples per second. Onboard storage permits data buffering of up to 256K samples, for all channels collectively, between the PCI bus and the cable interface. This allows the 16AO20 to sustain continuous throughput to the cable interface independent of the PCI bus interface. The 16AO20 also permits multiple boards to be synchronized so that all boards output data in unison.

1.6. Reference Material

The following reference material may be of particular benefit in using a 16AO20 board and this driver. The specifications provide the information necessary for an in depth understanding of the specialized features implemented on this board.

- The applicable *16AO20 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

2. Installation

2.1. CPU and Kernel Support

The driver is designed to operate with Linux kernel versions 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	x86	
		32-bit	64-bit
2.6.27	Red Hat Fedora Core 10	Yes	Yes
2.6.26	Red Hat Fedora Core 9 (with updates as of 9/26/2008)	Yes	Yes
2.6.25	Red Hat Fedora Core 9	Yes	Yes
2.6.23	Red Hat Fedora Core 8	Yes	Yes
2.6.21	Red Hat Fedora Core 7	Yes	Yes
2.6.18	Red Hat Fedora Core 6	Yes	Yes
2.6.16	SUSE 10.1	Yes	Yes
2.6.15	Red Hat Fedora Core 5	Yes	Yes
2.6.11	Red Hat Fedora Core 4	Yes	Yes
2.6.9	Red Hat Fedora Core 3	Yes	Yes
2.4.21	Red Hat Enterprise Linux Workstation Release 3	Yes	▨
2.4.20	Red Hat Linux 9	Yes	▨
2.4.18	Red Hat Linux 8.0	Yes	▨
2.4.7	Red Hat Linux 7.2	Yes	▨
2.2.14	Red Hat Linux 6.2	Yes	▨

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 has not been tested for SMP operation.

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/16ao20` file will be "`no`".

2.2. The /proc File System

While the driver is loaded, the text file `/proc/16ao20` can be read to obtain information about the driver. 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: 2.0.7
built: May 26 2009, 14:41:38
32-bit support: yes
boards: 1
models: 16AO20
```

Entry	Description
`version`	This gives the driver version number in the form `x.x.x`.
`built`	This gives the driver build date and time as a string. It is given in the C form of `printf("%s, %s", __DATE__, __TIME__)`.
`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.

2.3. File List

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

File	Description
`16ao20.linux.tar.gz`	This archive contains the driver, the samples and all related sources.
`16ao20_linux_um.pdf`	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 Structure	Content
`16ao20`	This is the source root directory. The user manual and overall make script are placed here.
`16ao20\aout`	This directory contains the Analog Output sample application. Refer to section 5.1 on page 16.
`16ao20\docsrc`	This directory contains the Document Source Code Examples. Refer to section 4 on page 15.
`16ao20\driver`	This directory contains the driver and its sources. Refer to section 3 on page 12.
`16ao20\fsamp`	This directory contains the FSAMP Calculator sample application. Refer to section 5.2 on page 17.
`16ao20\id`	This directory contains the Identification application. Refer to section 5.3 on page 18.
`16ao20\regs`	This directory contains the Register Access sample application. Refer to section 5.4 on page 18.
`16ao20\sbtest`	This directory contains the Single Board Test application. Refer to section 5.5 on page 19.
`16ao20\txrate`	This directory contains the Transmit Rate sample application. Refer to section 5.6 on page 20.
`16ao20\utils`	This directory contains utility sources used by the sample applications.

2.5. Installation

Install the driver and its related files following the below listed steps. This includes the device driver, the documentation source code, and the sample applications.

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 `16ao20.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 `16ao20` in the current directory, and then copies all of the archive's files into this new directory.

```
tar –xzvf 16ao20.linux.tar.gz
```

2.6. Removal

Follow the below steps to remove the driver and its related files. This includes the device driver, the documentation source code, and the sample applications.

1. Shutdown the driver as described in section 3.5 on page 14.

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 16ao20.linux.tar.gz 16ao20
   ```

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

   ```
   rm –rf /dev/16ao20*
   ```

5. If the automated startup procedure was adopted (see section 3.2.2 on page 13), then edit the system startup script `rc.local` and remove the line that invokes the 16AO20'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, and it will also load the driver. The script is named `make_all`. Follow the below steps to perform an overall make and to load the driver.

1. Change to the driver's directory, which may be `/usr/src/linux/drivers/16ao20`.

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

   ```
   ./make_all
   ```

3. The Driver

The driver and its related files are contained in the archive file `16ao20.linux.tar.gz`. The driver's files are summarized in the table below.

File	Description
`driver/*.c`	The driver source files.
`driver/*.h`	The driver header files.
`driver/start`	Shell script to install the driver executable and device nodes.
`driver/16ao20.h`	This is the main driver header file. This header should be included by 16AO20 applications.
`driver/Makefile`	The driver make file.

3.1. Build

NOTE: Building the driver requires installation of the kernel sources.

Follow the below steps to build the driver.

1. Change to the directory where the driver and its sources are installed, which may be `/usr/src/linux/drivers/16ao20/driver`.

2. Remove all existing build targets by issuing the below command.

```
make clean
```

3. Build the driver by issuing the below command.

```
make all
```

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.

3.2. 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 insure that the driver module in the install directory is the module that is loaded. This is accomplished by making sure that an already loaded module is first unloaded before attempting to load the module from the disk drive. In addition, the script also deletes and recreates the device nodes. This is done to insure 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 correspond to the number of boards successfully identified by the driver.

3.2.1. Manual Driver Startup Procedures

Start the driver manually by following the below listed steps.

1. Login as root user, as some of the steps require root privileges.

2. Change to the directory where the driver are installed, which may be `/usr/src/linux/drivers/16ao20/driver`.

3. 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 rebooted.

NOTE: The 16AO20 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.

4. Verify that the device driver module has been loaded by issuing the below command and examining the output. The module name `16ao20` should be included in the output.

   ```
   lsmod
   ```

5. 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/16ao20*
   ```

3.2.2. 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/16ao20/driver/start
   ```

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.

3.3. Verification

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

1. Verify that the file `/proc/16ao20` 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/16ao20
   ```

3.4. 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/16ao20` while the driver is loaded and running.

3.5. Shutdown

Shutdown the driver following the below listed steps.

1. Login as root user, as some of the steps require root privileges.
2. If the driver is currently loaded then issue the below command to unload the driver.

   ```
   rmmod 16ao20
   ```

3. Verify that the driver module has been unloaded by issuing the below command. The module name `16ao20` should not be in the listed output.

   ```
   lsmod
   ```

4. Document Source Code Examples

The archive file `16ao20.linux.tar.gz` contains all of the source code examples included in this document. In addition, the code is built into a statically linkable library usable with 16AO20 console applications. The library and sources are delivered undocumented and unsupported. 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. These files are located in the `docsrc` subdirectory under the 16AO20 root directory.

File	Description
`docsrc/*.c`	These are the C source files.
`docsrc/16ao20_dsl.h`	This is the library header file.
`docsrc/makefile`	This is the library make file.
`docsrc/makeile.dep`	This is an automatically generated make dependency file.

4.1. Build

Follow the below steps to compile the example files and build the library.

1. Change to the directory where the documentation sources are installed, which may be `/usr/src/linux/drivers/16ao20/docsrc`.

2. Remove all 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 all
   ```

4.2. Library Use

The library is used both at application compile time and at application link time. Compile time use has two requirements. First, include the header file `16ao20_dsl.h` in each module referencing a library component. Second, expand the include file search path to search the directory where the library header is located, which may be `/usr/src/linux/drivers/16ao20/docsrc`. Link time use also has two requirements. First, include the static library `16ao20_dsl.a` in the list of files to be linked into the application. Second, expand the library file search path to search the directory where the library is located, which may be `/usr/src/linux/drivers/16ao20/docsrc`.

5. Sample Applications

5.1. Analog Output - aout

This sample application provides a command line driven Linux application that configures a designated 16AO20 board and outputs test patterns on the first four channels. The application is provided without documentation or support, but it can be used as the starting point for application development on top of the 16AO20 Linux device driver. The application includes the below listed files.

File	Description
`aout/*.c`	These are the application's source files.
`aout/main.h`	This is the application's header file.
`aout/makefile`	This is the application make file.
`aout/makefile.dep`	This is an automatically generated make dependency file.
`docsrc/*`	These are utility sources used by the application.
`utils/*`	These are utility sources used by the application.

5.1.1. Build

Follow the below steps to build the sample application.

1. Change to the directory where the sample application is installed, which may be `/usr/src/linux/drivers/16ao20/aout`.

2. Remove all existing build targets by issuing the below command.

   ```
   make clean
   ```

3. Build the application by issuing the below command.

   ```
   make all
   ```

5.1.2. Execute

Follow the below steps to execute the sample application.

1. Change to the directory where the sample application is installed, which may be `/usr/src/linux/drivers/16ao20/aout`.

2. Start the sample application by issuing the command given below. Once started the application will configure the board, generate four test patterns then output the patterns to the first four channels. The patterns generated are a saw tooth wave with a falling slope, a saw tooth wave with a rising slope, a square wave with a 50% duty cycle and a sine wave. The pattern length is reported to the screen in both samples and duration. The duration of each invocation depends on the arguments specified, but should be a minimum of five to 10 seconds. The command line arguments are described in the table below.

   ```
   ./aout <-c> <-C> <-dma> <-m#> <-n#> <-p#> <-pio> <-r#> <index>
   ```

Argument	Description
`-c`	Repeat the operation until an error is encountered.
`-C`	Repeat the operation, but continue even if errors are encountered.
`-dma`	Use DMA when writing data.
`-m#`	When repeating the operation, stop after "#" minutes, where "#" is a decimal number.

`-n#`	When repeating the operation, stop after “#” interactions, where “#” is a decimal number.
`-p#`	This specifies the output period (or duration) in seconds, where “#” is a decimal number.
`-pio`	Use PIO when writing data.
`-r#`	This specifies the output data rate in samples per second, where “#” is a decimal number.
`index`	This is the index of the board to access.

5.2. FSAMP Calculator – fsamp

This sample console application provides a command line driven Linux utility to help users identify the ideal NRATE, NCLK and reference source options required for a desired sample rate, FSAMP. The application’s sources are summarized in the below table.

File	Description
`fsamp/*.c`	These are the application’s source files.
`fsamp/main.h`	This is the application’s header file.
`fsamp/makefile`	This is the application make file.
`fsamp/makefile.dep`	This is an automatically generated make dependency file.
`docsrc/*`	These are utility sources used by the application.
`utils/*`	These are utility sources used by the application.

5.2.1. Build

Follow the below steps to build the sample application.

1. Change to the directory where the sample application is installed, which may be `/usr/src/linux/drivers/16ao20/fsamp`.

2. Remove all existing build targets by issuing the below command.

```
make clean
```

3. Build the application by issuing the below command.

```
make all
```

5.2.2. Execute

Follow the below steps to execute the sample application.

1. Change to the directory where the sample application is installed, which may be `/usr/src/linux/drivers/16ao20/fsamp`.

2. Start the sample application by issuing the command given below. Once started the application will automatically calculate the NRATE, NCLK and oscillator reference selection values that produce the FSAMP value closest to that requested for the designated board. The calculation should complete is a second or less. The command line arguments are described in the table below.

```
./fsamp <rate> <index>
```

Argument	Description
`rate`	This is the desired sample rate.

`index`	This is the index of the board to access.

5.3. Identify Board - id

This sample console application provides a command line driven Linux application that provides 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. The application's sources are summarized in the below table.

File	Description
`id/*.c`	These are the application's source files.
`id/main.h`	This is the application's header file.
`id/makefile`	This is the application make file.
`id/makefile.dep`	This is an automatically generated make dependency file.
`docsrc/*`	These are utility sources used by the application.
`utils/*`	These are utility sources used by the application.

5.3.1. Build

Follow the below steps to build the sample application.

1. Change to the directory where the sample application is installed, which may be `/usr/src/linux/drivers/16ao20/id`.

2. Remove all existing build targets by issuing the below command.

   ```
   make clean
   ```

3. Build the application by issuing the below command.

   ```
   make all
   ```

5.3.2. Execute

Follow the below steps to execute the sample application.

1. Change to the directory where the sample application is installed, which may be `/usr/src/linux/drivers/16ao20/id`.

2. Start the sample application by issuing the command given below. Once started the application will automatically output identification information. A single iteration should take a second or less. The command line arguments are described in the table below.

   ```
   ./id <index>
   ```

Argument	Description
`index`	This is the index of the board to access.

5.4. Register Access - regs

This sample console application provides a menu based command line Linux application that permits interactive access to the board's registers, including write access to the GSC specific registers. The application's sources are summarized in the below table.

File	Description
`regs/*.c`	These are the application's source files.
`regs/main.h`	This is the application's header file.
`regs/makefile`	This is the application make file.
`regs/makefile.dep`	This is an automatically generated make dependency file.
`docsrc/*`	These are utility sources used by the application.
`utils/*`	These are utility sources used by the application.

5.4.1. Build

Follow the below steps to build the sample application.

1. Change to the directory where the sample application is installed, which may be `/usr/src/linux/drivers/16ao20/regs`.

2. Remove all existing build targets by issuing the below command.

   ```
   make clean
   ```

3. Build the application by issuing the below command.

   ```
   make all
   ```

5.4.2. Execute

Follow the below steps to execute the sample application.

1. Change to the directory where the sample application is installed, which may be `/usr/src/linux/drivers/16ao20/regs`.

2. Start the sample application by issuing the command given below. The command line argument is described in the table below.

   ```
   ./regs <index>
   ```

Argument	Description
`index`	This is the index of the board to access.

5.5. Single Board Test - sbtest

This sample console application provides a command line driven Linux application that tests the functionality of the driver and a specified board. The application's sources are summarized in the below table.

File	Description
`sbtest/*.c`	These are the application's source files.
`sbtest/main.h`	This is the application's header file.
`sbtest/makefile`	This is the application make file.
`sbtest/makefile.dep`	This is an automatically generated make dependency file.
`docsrc/*`	These are utility sources used by the application.
`utils/*`	These are utility sources used by the application.

5.5.1. Build

Follow the below steps to build the sample application.

1. Change to the directory where the sample application is installed, which may be `/usr/src/linux/drivers/16ao20/sbtest`.

2. Remove all existing build targets by issuing the below command.

```
make clean
```

3. Build the application by issuing the below command.

```
make all
```

5.5.2. Execute

NOTE: This application should be run with no cable attached.

Follow the below steps to execute the sample application.

1. Change to the directory where the sample application is installed, which may be `/usr/src/linux/drivers/16ao20/sbtest`.

2. Start the sample application by issuing the command given below. Once started the application will automatically performs a series of test operations. A single iteration may take up to 90 seconds to complete. The command line arguments are described in the table below.

```
./sbtest <-c> <-C> <-m#> <-n#> <index>
```

Argument	Description
`-c`	Repeat the operation until an error is encountered.
`-C`	Repeat the operation, but continue even if errors are encountered.
`-m#`	When repeating the operation, stop after “#” minutes, where “#” is a decimal number.
`-n#`	When repeating the operation, stop after “#” interactions, where “#” is a decimal number.
`index`	This is the index of the board to access.

5.6. Transmit Rate - txrate

This sample console application provides a command line driven Linux application that will write a specified amount of data to a specified board. The data written is a fixed pattern. The application’s sources are summarized in the below table.

File	Description
`txrate/*.c`	These are the application’s source files.
`txrate/main.h`	This is the application’s header file.
`txrate/makefile`	This is the application make file.
`txrate/makefile.dep`	This is an automatically generated make dependency file.
`docsrc/*`	These are utility sources used by the application.
`utils/*`	These are utility sources used by the application.

5.6.1. Build

Follow the below steps to build the sample application.

1. Change to the directory where the sample application is installed, which may be `/usr/src/linux/drivers/16ao20/txrate`.

2. Remove all existing build targets by issuing the below command.

   ```
   make clean
   ```

3. Build the application by issuing the below command.

   ```
   make all
   ```

5.6.2. Execute

Follow the below steps to execute the sample application.

1. Change to the directory where the sample application is installed, which may be `/usr/src/linux/drivers/16ao20/txrate`.

2. Start the sample application by issuing the command given below. Once started the application will automatically perform a series of operations. With the default options a single iteration should take less than 15 seconds to complete. The command line arguments are described in the table below.

   ```
   ./txrate <-c> <-C> <-dma> <-m#> <-n#> <-pio> <-r#> <index>
   ```

Argument	Description
`-c`	Repeat the operation until an error is encountered.
`-C`	Repeat the operation, but continue even if errors are encountered.
`-dma`	Perform the transfer using DMA.
`-m#`	When repeating the operation, stop after "#" minutes, where "#" is a decimal number.
`-n#`	When repeating the operation, stop after "#" interactions, where "#" is a decimal number.
`-pio`	Perform the transfer using PIO. This is the default/
`-r#`	This specifies the number of megabytes to write, where "#" is a decimal number.
`index`	This is the index of the board to access.

6. Driver Interface

The 16AO20 driver conforms to the device driver standards required by the Linux Operating System and contains the standard driver entry points. The device driver provides a standard driver interface to 16AO20 boards for Linux applications. The interface includes various macros, data types and functions, all of which are described in the following paragraphs. The 16AO20 specific portion of the driver interface is defined in the header file `16ao20.h`, portions of which are described in this section. The header defines numerous items in addition to those described here.

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

6.1. Macros

The driver interface includes the following macros, which are defined in `16ao20.h`. The header also contains various other utility type macros, which are provided without documentation.

6.1.1. IOCTL

The IOCTL macros are documented in section 6.3 beginning on page 26.

6.1.2. Registers

The following gives the complete set of 16AO20 registers.

6.1.2.1. GSC Registers

The following table gives the complete set of GSC specific 16AO20 registers. For the set of supported registers and detailed definitions of these registers please refer to the appropriate *16AO20 User Manual*.

Macro	Description
`AO20_GSC_ACR`	Adjustable Clock Register
`AO20_GSC_AVR`	Autocal Values Register
`AO20_GSC_BCR`	Board Control Register
`AO20_GSC_BOR`	Buffer Operations Register
`AO20_GSC_CSR`	Channel Selection Register
`AO20_GSC_FOR`	Firmware Options Register
`AO20_GSC_ODBR`	Output Data Buffer Register
`AO20_GSC_SRR`	Sample Rate Register

6.1.2.2. PCI Configuration Registers

Access to the PCI registers a 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 `16ao20.h`.

6.1.2.3. PLX PCI9080 Feature Set Registers

Access to the PLX registers a 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 `16ao20.h`.

6.2. Functions

The driver interface includes the following functions.

6.2.1. close()

This function is the entry point to close a connection to an open 16AO20 board.

Prototype

```
int close(int fd);
```

Argument	Description
`fd`	This is the file descriptor of the device to be closed.

Return Value	Description
`-1`	An error occurred. Consult `errno`.
`0`	The operation succeeded.

Example

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

#include "16ao20_dsl.h"

int ao20_dsl_close(int fd)
{
    int err;
    int status;

    status  = close(fd);

    if (status == -1)
        printf("ERROR: close() failure, errno = %d\n", errno);

    err = (status == -1) ? 1 : 0;
    return(err);
}
```

6.2.2. ioctl()

This function is the entry point to performing setup and control operations on a 16AO20 board. 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 any additional arguments. The set of supported IOCTL services is defined in section 6.3 beginning on page 26.

Prototype

```
int ioctl(int fd, int request, ...);
```

Argument	Description
`fd`	This is the file descriptor of the device to access.
`request`	This specifies the desired operation to be performed.

`...`	This is any additional arguments. If `request` does not call for any additional arguments, then any additional arguments provided are ignored. The 16AO20 IOCTL services use at most one argument.

Return Value	Description
`-1`	An error occurred. Consult `errno`.
`0`	The operation succeeded.

Example

```
#include <errno.h>
#include <stdio.h>
#include <sys/ioctl.h>

#include "16ao20_dsl.h"

int ao20_dsl_ioctl(int fd, int request, void *arg)
{
    int err;
    int status;

    status  = ioctl(fd, request, arg);

    if (status == -1)
        printf("ERROR: ioctl() failure, errno = %d\n", errno);

    err = (status == -1) ? 1 : 0;
    return(err);
}
```

6.2.3. open()

This function is the entry point to open a connection to a 16AO20 board. The pathname to a 16AO20 device node is `/dev/16ao20`*`n`*, where the trailing "`n`" is the zero based index of the board to access.

Prototype

```
int open(const char* pathname, int flags);
```

Argument	Description
`pathname`	This is the name of the device to open.
`flags`	This is the desired read/write access. Use `O_RDWR`.

NOTE: Another form of the `open()` function has a `mode` argument. This form is not displayed here as the `mode` argument is ignored when opening an existing file/device.

Return Value	Description
`-1`	An error occurred. Consult `errno`.
else	A valid file descriptor.

Example

```
#include <errno.h>
#include <fcntl.h>
```

```
#include <stdio.h>

#include "16ao20_dsl.h"

int ao20_dsl_open(unsigned int board)
{
    int     fd;
    char    name[80];

    sprintf(name, AO20_DEV_BASE_NAME "%u", board);
    fd  = open(name, O_RDWR);

    if (fd == -1)
    {
        printf( "ERROR: open() failure on %s, errno = %d\n",
                name,
                errno);
    }

    return(fd);
}
```

6.2.4. read()

This service is not implemented as the 16AO20 has no data collection or data reception capabilities. This service will always return an error.

6.2.5. write()

This function is the entry point to writing data to an open 16AO20. This function should only be called after a successful open of the respective device. The function writes up to `count` bytes to the board. The return value is the number of bytes actually written.

> **NOTE:** Applications may experience improved responsiveness with read requests by coordinating the Buffer Size setting with the number of samples to write. Refer to the `AO20_IOCTL_BUFFER_SIZE` service of section 6.3.6 on page 28.

Prototype

```
int write(int fd, const void *buf, size_t count);
```

Argument	**Description**
`fd`	This is the file descriptor of the device to access.
`buf`	The data to write comes from here.
`count`	This is the desired number of bytes to write. This must be a multiple of four (4).

Return Value	**Description**
`-1`	An error occurred. Consult `errno`.
`0` to `count`	The operation succeeded. For blocking I/O a return value less than `count` indicates that the request timed out. For non-blocking I/O a return value less than `count` indicates that the operation ended prematurely.

Example

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

#include "16ao20_dsl.h"

int ao20_dsl_write(int fd, const __u32 *buf, size_t samples)
{
    size_t  bytes;
    int     status;

    bytes   = samples * 4;
    status  = write(fd, buf, bytes);

    if (status == -1)
        printf("ERROR: write() failure, errno = %d\n", errno);
    else
        status  /= 4;

    return(status);
}
```

6.3. IOCTL Services

The 16AO20 driver implements the following IOCTL services. Each service is described along with the applicable `ioctl()` function arguments. In the definitions given the optional argument is identified as `arg`. Unless otherwise stated the return value definitions are those defined for the `ioctl()` function call and any error codes are accessed via `errno`.

6.3.1. AO20_IOCTL_AUTO_CALIBRATE

This service initiates an auto-calibration cycle. Most configuration settings should be made before running an auto-calibration cycle. The driver waits for the operation to complete before returning.

NOTE: Do not access the board while an auto-calibration cycle is in progress. Doing so may produce indeterminate results, and may lockup the board.

Usage

`ioctl()` Argument	Description
`request`	`AO20_IOCTL_AUTO_CALIBRATE`
`arg`	Not used.

6.3.2. AO20_IOCTL_BUFFER_CLEAR

This service immediately clears the current content from the output buffer. It also clears the board’s buffer overrun and frame overrun status. This service does not halt data output.

Usage

`ioctl()` Argument	Description
`request`	`AO20_IOCTL_BUFFER_CLEAR`
`arg`	Not used.

6.3.3. AO20_IOCTL_BUFFER_MODE

This service configures the board's handling of data once it leaves the output buffer.

Usage

`ioctl()` Argument	Description
`request`	`AO20_IOCTL_BUFFER_MODE`
`arg`	`__s32*`

Valid argument values are as follows.

Value	Description
`-1`	Retrieve the current setting.
`AO20_BUFFER_MODE_CIRC`	Buffer data is recycled when it exits the buffer.
`AO20_BUFFER_MODE_OPEN`	The buffer data is not recycled when it exits the buffer.

6.3.4. AO20_IOCTL_BUFFER_OVER_DATA

This service operates on the Buffer Overflow status.

Usage

`ioctl()` Argument	Description
`request`	`AO20_IOCTL_BUFFER_OVER_DATA`
`arg`	`__s32*`

Valid argument values are as follows.

Value	Description
`AO20_BUFFER_OVER_DATA_CHK`	Report if an overflow has occurred.
`AO20_BUFFER_OVER_DATA_CLR`	Clear the overflow status.

The following values are those returned when checking on the overflow status.

Value	Description
`AO20_BUFFER_OVER_DATA_NO`	An overflow did not occur.
`AO20_BUFFER_OVER_DATA_YES`	An overflow did occur.

6.3.5. AO20_IOCTL_BUFFER_OVER_FRAME

This service operates on the Frame Overflow status.

Usage

`ioctl()` Argument	Description
`request`	`AO20_IOCTL_BUFFER_OVER_FRAME`
`arg`	`__s32*`

Valid argument values are as follows.

Value	Description
`AO20_BUFFER_OVER_FRAME_CHK`	Report if an overflow has occurred.
`AO20_BUFFER_OVER_FRAME_CLR`	Clear the overflow status.

The following values are those returned when checking on the overflow status.

Value	Description
AO20_BUFFER_OVER_FRAME_NO	An overflow did not occur.
AO20_BUFFER_OVER_FRAME_YES	An overflow did occur.

6.3.6. AO20_IOCTL_BUFFER_SIZE

This service configures the active size of the output buffer.

Usage

`ioctl()` Argument	Description
request	AO20_IOCTL_BUFFER_SIZE
arg	__s32*

Valid argument values are as follows.

Value	Description
-1	Retrieve the current setting.
AO20_BUFFER_SIZE_8	Set the buffer's active size to 8 samples.
AO20_BUFFER_SIZE_16	Set the buffer's active size to 16 samples.
AO20_BUFFER_SIZE_32	Set the buffer's active size to 32 samples.
AO20_BUFFER_SIZE_64	Set the buffer's active size to 64 samples.
AO20_BUFFER_SIZE_128	Set the buffer's active size to 128 samples.
AO20_BUFFER_SIZE_256	Set the buffer's active size to 256 samples.
AO20_BUFFER_SIZE_512	Set the buffer's active size to 512 samples.
AO20_BUFFER_SIZE_1K	Set the buffer's active size to 1K samples (1,024).
AO20_BUFFER_SIZE_2K	Set the buffer's active size to 2K samples (2,048).
AO20_BUFFER_SIZE_4K	Set the buffer's active size to 4K samples (4,096).
AO20_BUFFER_SIZE_8K	Set the buffer's active size to 8K samples (8,192).
AO20_BUFFER_SIZE_16K	Set the buffer's active size to 16K samples (16,384).
AO20_BUFFER_SIZE_32K	Set the buffer's active size to 32K samples (32,768).
AO20_BUFFER_SIZE_64K	Set the buffer's active size to 64K samples (65,536).
AO20_BUFFER_SIZE_128K	Set the buffer's active size to 128K samples (131,072).
AO20_BUFFER_SIZE_256K	Set the buffer's active size to 256K samples (262,144).

6.3.7. AO20_IOCTL_BUFFER_STATUS

This service reports the relative fill level of the active buffer. The buffer's active size is set with the `AO20_IOCTL_BUFFER_SIZE` service (see section 6.3.6 on page 28).

Usage

`ioctl()` Argument	Description
request	AO20_IOCTL_BUFFER_STATUS
arg	__s32*

The service returns one of the following values.

Value	Description
AO20_BUFFER_STATUS_EMPTY	The buffer is empty.
AO20_BUFFER_STATUS_1Q_FULL	The buffer is less than ¼ full.
AO20_BUFFER_STATUS_MEDIUM	The buffer is from ¼ to ¾ full.

AO20_BUFFER_STATUS_3Q_FULL	The buffer is ¾ full or more.
AO20_BUFFER_STATUS_FULL	The buffer is full.

6.3.8. AO20_IOCTL_BURST_READY

This service reports the board's readiness for burst initiation.

Usage

`ioctl()` Argument	Description
request	AO20_IOCTL_BURST_READY
arg	__s32*

The service returns one of the following values.

Value	Description
AO20_BURST_READY_NO	The board is not ready for burst initiation.
AO20_BURST_READY_YES	The board is ready for burst initiation.

6.3.9. AO20_IOCTL_BURST_TRIG_SRC

This service controls the trigger source selection for triggered burst operation.

Usage

`ioctl()` Argument	Description
request	AO20_IOCTL_BURST_TRIG_SRC
arg	__s32*

Valid argument values are as follows.

Value	Description
-1	Retrieve the current setting.
AO20_BURST_TRIG_SRC_EXT	Utilize external burst triggering.
AO20_BURST_TRIG_SRC_SW	Utilize software burst triggering.

6.3.10. AO20_IOCTL_BURST_TRIGGER

This service initiates an output burst cycle. The service waits for up to the write timeout period for the operation to complete. (See `AO20_IOCTL_TX_IO_TIMEOUT`, section 6.3.36, page 39.)

Usage

`ioctl()` Argument	Description
request	AO20_IOCTL_BURST_TRIGGER
arg	Not used.

6.3.11. AO20_IOCTL_CHANNEL_SEL

This service enables or disables channels for outputting data.

Usage

`ioctl()` Argument	Description
request	AO20_IOCTL_CHANNEL_SEL
arg	__s32*

Valid argument values are any valid set of bits for the set of supported channels, and `-1`. If a bit is set, then the corresponding channel is enabled. A zero bit disables the channel. The lowest significant bit corresponds to channel zero. The value `-1` is used to retrieve the current setting.

6.3.12. AO20_IOCTL_CLOCK_ENABLE

This service enables and disables clocking of output data.

Usage

`ioctl()` Argument	Description
request	AO20_IOCTL_CLOCK_ENABLE
arg	__s32*

Valid argument values are as follows.

Value	Description
-1	Retrieve the current setting.
AO20_CLOCK_ENABLE_NO	This disables the output sample clock.
AO20_CLOCK_ENABLE_YES	This enables the output sample clock.

6.3.13. AO20_IOCTL_CLOCK_OSC_SRC

This service selects the rate generator's clock source.

Usage

`ioctl()` Argument	Description
request	AO20_IOCTL_CLOCK_OSC_SRC
arg	__s32*

Valid argument values are as follows.

Value	Description
-1	Retrieve the current setting.
AO20_CLOCK_OSC_SRC_ALT	This selects the alternate source, which is generally derived from the 16MHz source. *
AO20_CLOCK_OSC_SRC_PRI	This selects the primary source, which generally refers to the 30MHz oscillator. *

* Refer to the hardware reference manual for additional information.

6.3.14. AO20_IOCTL_CLOCK_READY

This service reports the board's readiness to accept a software or external clock.

Usage

`ioctl()` Argument	Description
`request`	`AO20_IOCTL_CLOCK_READY`
`arg`	`__s32*`

The service returns one of the following values.

Value	Description
`AO20_CLOCK_READY_NO`	The board is not ready for a clock.
`AO20_CLOCK_READY_YES`	The board is ready for a clock.

6.3.15. AO20_IOCTL_CLOCK_SRC

This service selects the source for the output sample clock.

Usage

`ioctl()` Argument	Description
`request`	`AO20_IOCTL_CLOCK_SRC`
`arg`	`__s32*`

Valid argument values are as follows.

Value	Description
`-1`	Retrieve the current setting.
`AO20_CLOCK_SRC_EXT_SW`	This selects the external/software option as the clock source.
`AO20_CLOCK_SRC_INT`	This selects the internal rate generator as the source.

6.3.16. AO20_IOCTL_CLOCK_SW

This service initiates an output clock cycle. The service waits for up to the write timeout period for the operation to complete.

Usage

`ioctl()` Argument	Description
`request`	`AO20_IOCTL_CLOCK_SW`
`arg`	Not used.

6.3.17. AO20_IOCTL_DATA_FORMAT

This service sets the data encoding format.

Usage

`ioctl()` Argument	Description
`request`	`AO20_IOCTL_DATA_FORMAT`
`arg`	`__s32*`

Valid argument values are as follows.

Value	Description
`-1`	Retrieve the current setting.

AO20_DATA_FORMAT_2S_COMP	Select the Twos Compliment data format.
AO20_DATA_FORMAT_OFF_BIN	Select the Offset Binary encoding format.

6.3.18. AO20_IOCTL_GROUND_SENSE

This service configures the board's ground sense logic.

Usage

ioctl() Argument	**Description**
request	AO20_IOCTL_GROUND_SENSE
arg	__s32*

Valid argument values are as follows.

Value	**Description**
-1	Retrieve the current setting.
AO20_GROUND_SENSE_DISABLE	This disables remote ground sensing.
AO20_GROUND_SENSE_REMOTE	This selects remote ground sensing.

6.3.19. AO20_IOCTL_INITIALIZE

This service returns the board to its initialized state. The initialize operation sets all hardware settings to their defaults. The driver waits for the operation to complete before returning.

Usage

ioctl() Argument	**Description**
request	AO20_IOCTL_INITIALIZE
arg	Not used.

6.3.20. AO20_IOCTL_IRQ_ENABLE

This service enables or disables firmware interrupts, which is the master for the firmware interrupts.

Usage

ioctl() Argument	**Description**
request	AO20_IOCTL_IRQ_ENABLE
arg	__s32*

Valid argument values are as follows.

Value	**Description**
-1	Retrieve the current status.
AO20_IRQ_ENABLE_NO	This option prevents the firmware interrupt from being passed to the processor. The interrupt is still operational, but it will not result in the processor being interrupted.
AO20_IRQ_ENABLE_YES	This option permits firmware interrupts to be passed to the processor.

6.3.21. AO20_IOCTL_IRQ_SEL

This service configures the interrupt source selection for the firmware interrupt.

Usage

ioctl() Argument	Description
request	AO20_IOCTL_IRQ_SEL
arg	__s32*

Valid argument values are as follows.

Value	Description
-1	Retrieve the current setting.
AO20_IRQ_SEL_AUTOCAL_DONE	This refers to the completion of auto-calibration.
AO20_IRQ_SEL_BUF_1Q_FULL	The refers to the buffer becoming less than ¼ full.
AO20_IRQ_SEL_BUF_3Q_FULL	The refers to the buffer becoming more than 3/4 full.
AO20_IRQ_SEL_BUF_EMPTY	The refers to the buffer becoming empty.
AO20_IRQ_SEL_BURST_TRIG_READY	This refers to the board becoming ready for a burst trigger.
AO20_IRQ_SEL_INIT_DONE	This refers to the completion of initialization.
AO20_IRQ_SEL_LOAD_READY	This refers to a circular buffer becoming ready to receive data.
AO20_IRQ_SEL_LOAD_READY_END	This refers to a circular buffer becoming not ready to receive data.

6.3.22. AO20_IOCTL_IRQ_STATE

This service deals with the status of the firmware interrupt.

Usage

ioctl() Argument	Description
request	AO20_IOCTL_IRQ_STATE
arg	__s32*

Valid argument values are as follows.

Value	Description
-1	Retrieve the current state.
AO20_IRQ_STATE_CLEAR	This option will clear the interrupt status if it is set.
AO20_IRQ_STATE_IGNORE	This option takes no action regarding the current status.

When retrieving the current state the below values are returned.

Value	Description
AO20_IRQ_STATE_ACTIVE	An interrupt has been generated.
AO20_IRQ_STATE_IDLE	An interrupt has not been generated.

6.3.23. AO20_IOCTL_LOAD_READY

This service reports the buffer’s readiness to receive additional data when in circular buffer mode.

Usage

ioctl() Argument	Description
request	AO20_IOCTL_LOAD_READY
arg	__s32*

Valid values returned by the service are as follows.

Value	**Description**
`AO20_LOAD_READY_NO`	The buffer is not ready to receive additional data.
`AO20_LOAD_READY_YES`	The buffer is ready to receive additional data.

6.3.24. AO20_IOCTL_LOAD_REQUEST

This service requests that buffer become ready to receive additional data when in circular buffer mode. The service waits for up to the write timeout period for the operation to complete.

Usage

`ioctl()` Argument	**Description**
`request`	`AO20_IOCTL_LOAD_REQUEST`
`arg`	Not used.

6.3.25. AO20_IOCTL_NCLK

This service sets the adjustable clock's NCLK adjustment value.

Usage

`ioctl()` Argument	**Description**
`request`	`AO20_IOCTL_NCLK`
`arg`	`__s32*`

Valid argument values are in the range from `0` to `0x1FF`, and `-1`. The value `-1` is used to retrieve the current setting.

6.3.26. AO20_IOCTL_NRATE

This service sets the rate divider's NRATE adjustment value.

Usage

`ioctl()` Argument	**Description**
`request`	`AO20_IOCTL_NRATE`
`arg`	`__s32*`

Valid argument values are in the range from `68` to `0xFFFF`, and `-1`. The value `-1` is used to retrieve the current setting.

6.3.27. AO20_IOCTL_OUTPUT_MODE

This service configures the buffer's data output mode.

Usage

`ioctl()` Argument	**Description**
`request`	`AO20_IOCTL_OUTPUT_MODE`
`arg`	`__s32*`

Valid argument values are as follows.

Value	Description
`-1`	Retrieve the current setting.
`AO20_BURST_OUTPUT_MODE_SEQ`	Channel data is output to one channel at a time, sequentially. *
`AO20_BURST_OUTPUT_MODE_SIM`	Channel data is output to all channels simultaneously.

* Refer to the hardware reference manual to see how this affects the sample rate.

6.3.28. AO20_IOCTL_QUERY

This service queries the driver for various pieces of information about the board and the driver.

Usage

`ioctl()` Argument	Description
`request`	`AO20_IOCTL_QUERY`
`arg`	`__s32*`

Valid argument values are as follows.

Value	Description
`AO20_QUERY_AUTO_CAL_MS`	This returns the maximum duration of the Auto Calibration cycle in milliseconds.
`AO20_QUERY_CHANNEL_MASK`	This is the valid mask for all selectable channels and is based on the number of channels the board supports.
`AO20_QUERY_CHANNEL_MAX`	This returns the maximum number of output channels supported by all boards of the same model as the board accessed.
`AO20_QUERY_CHANNEL_QTY`	This returns the actual number of output channels on the current board.
`AO20_QUERY_COUNT`	This returns the number of query options supported by the IOCTL service.
`AO20_QUERY_DEVICE_TYPE`	This returns the identifier value for the board’s type. The value is a member of the `gsc_dev_type_t` enumeration, which is defined in `gsc_common.h`.
`AO20_QUERY_FIFO_SIZE`	This returns the size of the output buffer in samples.
`AO20_QUERY_FILTER`	This returns the identifier for the installed filter option. Refer to the table below for the returned option values.
`AO20_QUERY_FREF_DEFAULT`	This gives the default FREF value in hertz.
`AO20_QUERY_FSAMP_MAX`	This gives the maximum sample rate in S/S.
`AO20_QUERY_FSAMP_MIN`	This gives the minimum sample rate in S/S.
`AO20_QUERY_INIT_MS`	This returns the duration of a board initialization in milliseconds.
`AO20_QUERY_LAST`	This is included for reference only and should not be used by applications. Applications should use the COUNT option instead.
`AO20_QUERY_NCLK_MASK`	This returns the mask for the adjustable clock’s NCLK value.
`AO20_QUERY_NCLK_MAX`	This returns the maximum supported NCLK value.
`AO20_QUERY_NCLK_MIN`	This returns the minimum supported NCLK value.
`AO20_QUERY_NRATE_MASK`	This returns the mask for the board’s NRATE field.
`AO20_QUERY_NRATE_MAX`	This returns the maximum supported NRATE value.
`AO20_QUERY_NRATE_MIN`	This returns the minimum supported NRATE value.
`AO20_QUERY_RANGE`	This returns the identifier for the installed voltage range option. Refer to the table below for the returned option values.

Valid return values for the `AO20_QUERY_FILTER` option are as follows.

Value	Description
`AO20_FILTER_NONE`	No filter is installed.

AO20_FILTER_10KHZ	A 10KHz filter is installed.
AO20_FILTER_100KHZ	A 100KHz filter is installed.

Valid return values for the `AO20_QUERY_RANGE` option are as follows.

Value	Description
AO20_RANGE_2_5	The board is hardwired for the voltage range of ±2.5V.
AO20_RANGE_5	The board is hardwired for the voltage range of ±5V.
AO20_RANGE_10	The board is hardwired for the voltage range of ±10V.

Valid return values are as indicated in the above tables and as given in the below table.

Value	Description
AO20_IOCTL_QUERY_ERROR	Either there was a processing error or the query option is unrecognized.

6.3.29. AO20_IOCTL_REG_MOD

This service performs a read-modify-write of a 16AO20 register. This includes only the GSC firmware registers. The PCI and PLX Feature Set Registers are read-only. Refer to `16ao20.h` for a complete list of the GSC firmware registers.

Usage

ioctl() Argument	Description
request	AO20_IOCTL_REG_MOD
arg	gsc_reg_t*

Definition

```
typedef struct
{
    __u32   reg;
    __u32   value;
    __u32   mask;
} gsc_reg_t;
```

Fields	Description
reg	This is set to the identifier for the register to access.
value	This contains the value for the register bits to modify.
mask	This specifies the set of bits to modify. If a bit here is set, then the respective register bits is modified. If a bit here is zero, then the respective register bit is unmodified.

6.3.30. AO20_IOCTL_REG_READ

This service reads the value of a 16AO20 register. This includes the PCI registers, the PLX Feature Set Registers and the GSC firmware registers. Refer to `16ao20.h` and `gsc_pci9080.h` for the complete list of accessible registers.

Usage

ioctl() Argument	Description
request	AO20_IOCTL_REG_READ
arg	gsc_reg_t*

Definition

```
typedef struct
{
    __u32   reg;
    __u32   value;
    __u32   mask;
} gsc_reg_t;
```

Fields	Description
`reg`	This is set to the identifier for the register to access.
`value`	This is the value read from the specified register.
`mask`	This is ignored for read request.

6.3.31. AO20_IOCTL_REG_WRITE

This service writes a value to a 16AO20 register. This includes only the GSC firmware registers. The PCI and PLX Feature Set Registers are read-only. Refer to `16ao20.h` for a complete list of the GSC firmware registers.

Usage

`ioctl()` Argument	Description
`request`	`AO20_IOCTL_REG_WRITE`
`arg`	`gsc_reg_t*`

Definition

```
typedef struct
{
    __u32   reg;
    __u32   value;
    __u32   mask;
} 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 specified register.
`mask`	This is ignored for write request.

6.3.32. AO20_IOCTL_SAMPLE_MODE

This service controls how and when data is output to the cable interface.

Usage

`ioctl()` Argument	Description
`request`	`AO20_IOCTL_SAMPLE_MODE`
`arg`	`__s32*`

Valid argument values are as follows.

Value	Description
`-1`	Retrieve the current setting.
`AO20_SAMPLE_MODE_BURST`	This selects triggered burst operation.

AO20_SAMPLE_MODE_CONT	This selects continuous output operation.

6.3.33. AO20_IOCTL_TX_IO_MODE

This service sets the I/O mode used for data write requests.

> **NOTE:** Applications may experience improved responsiveness with write requests by coordinating the Active Buffer Size with the number of samples in the write request. Refer to the AO20_IOCTL_BUFFER_SIZE service of section 6.3.6 on page 28.

Usage

ioctl() Argument	**Description**
request	AO20_IOCTL_TX_IO_MODE
arg	__s32*

Valid argument values are as follows.

Value	**Description**
-1	Retrieve the current setting.
GSC_IO_MODE_DMA	Use DMA.
GSC_IO_MODE_PIO	Use PIO mode, which is repetitive register access. This is the default.

6.3.34. AO20_IOCTL_TX_IO_OVER_DATA

This service configures the write service to check for an output buffer data overflow before performing write operations. Sample data is lost when there is a buffer overflow

Usage

ioctl() Argument	**Description**
request	AO20_IOCTL_TX_IO_OVER_DATA
arg	__s32*

Valid argument values are as follows.

Value	**Description**
-1	Retrieve the current setting.
AO20_TX_IO_OVER_DATA_CHECK	Perform the check. This is the default.
AO20_TX_IO_OVER_DATA_IGNORE	Do not perform the check.

6.3.35. AO20_IOCTL_TX_IO_OVER_FRAME

This service configures the write service to check for a frame overflow before performing write operations. Sample data is lost when there is a frame overflow

Usage

ioctl() Argument	**Description**
request	AO20_IOCTL_TX_IO_OVER_FRAME
arg	__s32*

Valid argument values are as follows.

Value	Description
`-1`	Retrieve the current setting.
`AO20_TX_IO_OVER_FRAME_CHECK`	Perform the check. This is the default.
`AO20_TX_IO_OVER_FRAME_IGNORE`	Do not perform the check.

6.3.36. AO20_IOCTL_TX_IO_TIMEOUT

This service sets the timeout limit for data write requests. The value is expressed in seconds.

Usage

`ioctl()` Argument	Description
`request`	`AO20_IOCTL_TX_IO_TIMEOUT`
`arg`	`__s32*`

Valid argument values are in the range from zero to `3600`, and `-1`. A value of zero tells the driver not to sleep in order to wait for more buffer space to become available, and should only be used with PIO mode writes. A value of `-1` is used to retrieve the current setting.

6.3.37. AO20_IOCTL_XCVR_TYPE

This service selects TTL or LVDS signaling on the external digital cable signals.

Usage

`ioctl()` Argument	Description
`request`	`AO20_IOCTL_XCVR_TYPE`
`arg`	`__s32*`

Valid argument values are as follows.

Value	Description
`-1`	Retrieve the current setting.
`AO20_XCVR_TYPE_LVDS`	Use LVDS signaling.
`AO20_XCVR_TYPE_TTL`	Use TTL signaling.

7. Operation

This section explains some operational procedures using the board. This is in no way intended to be a comprehensive guide on using the 16AO20. This is simply to address a very few issues relating to the board's use.

7.1. Data Output

Data output is essentially a three-step process. A simplified version of this process is outlined below.

> **NOTE:** These steps are guidelines only. The actual steps needed may vary significantly from one application to another.

1. Initialize the board to put the 16AO20 in a known state.
2. Perform the steps required for any desired configuration, buffer size, active channels, etc.
3. Write the data as it becomes available according to the application's needs.

7.1.1.1. PIO

This is called Programmed I/O and involves repetitive register accesses. In this mode the driver will write data to the output buffer one value at a time. As needed, the driver will repeatedly sleep for one system time tick in order to wait for addition space in the output buffer. This process is repeated until the data is exhausted or the I/O timeout expires, whichever occurs first.

7.1.1.2. DMA

For DMA transfers, hardware onboard the 16AO20 is used to transfer the data without processor intervention. In this mode the driver checks for available space in the output buffer. When sufficient space is available a DMA transfer is performed. Depending on the size of the write request, the driver may break the request into smaller transfers in order to insure data integrity. The breakup is based on the size of the request relative to the size of the active buffer. If the active buffer is empty, then the driver will perform a DMA transfer for up to the size of the active buffer. If the active buffer is up to ¼ full, then the driver will perform a DMA transfer for up to ¾ the size of the active buffer. If the active buffer is from ¼ to ¾ full, then the driver will perform a DMA transfer for up to the ¼ the size of the active buffer. If the active buffer is ¾ full or more, then the driver will sleep for one system timer tick and then check again. The process is repeated until the data is exhausted or the I/O timeout expires, whichever occurs first.

Document History

Revision	Description
May 26, 2009	Updated to release 2.0.7.1.
May 19, 2009	This is the initial release of the 2.x driver.