16AO12

16-Bit, 12 Channel High-Speed Analog Output Board

PMC-16AO12

LINUX Device Driver User Manual

Manual Revision: April 18, 2006

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

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 material, 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 to ECO control, **General Standards Corporation** assumes no responsibility for any errors that may exist in this document. No commitment is made to update or keep current the information contained in this document.

General Standards Corporation does not assume any liability arising out of the application or use of any 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 product to improve reliability, performance, function, or design.

The Purchaser of this software may use or modify in source form the subject software, but not to re-market or distribute it to outside agencies or separate internal company divisions. The software, however, may be embedded in the Purchaser's distributed software. In the event the Purchaser's customers require the software source code, then they would have to purchase their own copy of the software.

General Standards Corporation makes no warranty of any kind with regard to this software, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose and makes this software available solely on an "as-is" basis. **General Standards Corporation** reserves the right to make changes in this software without reservation and without notification to its users.

The information in this document is subject to change without notice. This document may be copied or reproduced provided it is in support of products from **General Standards Corporation**. For any other use, no part of this document may be copied or reproduced in any form or by any means without prior written consent of **General Standards Corporation.**

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

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

Table of Contents

1. Introduction

1.1. Purpose

The purpose of this document is to describe the interface to the 16AO12 LINUX device driver. This software provides the interface between application software and the 16AO12 board.

1.2. Acronyms

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

Acronyms	Description
DMA	Direct Memory Access
PCI	Peripheral Component Interconnect

1.3. Definitions

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

Term	Definition
Driver	Driver means the kernel mode device driver, which runs in the kernel space with kernel mode privileges.
Application	Application means the user mode process, which runs in the user space with user mode privileges.

1.4. Software Overview

The 16AO12 driver software executes under control of the Linux operating system. The device driver allows the user to open and close one or more 16AO12 boards, then read and write data to the registers on the hardware.

1.5. Hardware Overview

The 16AO12 is a 16-Bit, 12 channel high-speed analog output board. See the hardware manual for a detailed description of the hardware features.

1.6. Reference Material

The following reference material may be of particular benefit in using the 16AO12 and this driver. The specifications provide the information necessary for an in depth understanding of the specialized features implemented on this board.

- The applicable *16AO12 Reference 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 a Linux kernel-mode module. The driver has been tested on a PC system with dual Intel x86 processors running kernel 2.4.18-14SMP, and 2.6.8-1.521smp. Support for the 2.2 kernel has been left in the driver, but has not been built or tested. The driver will have to be rebuilt for the version of the kernel running on the target machine.

2.2. The Driver

This driver and its related files are:

File	Description
`*.c`	The driver source files.
`plx_regs.h`	Defines for the PLX chip, used internally by the driver.
`README.TXT`	Revision history and late-breaking news.
`Internals.h`	Internal defines for the driver.
`gsc16ao12_ioctl.h`	Defines common to the driver and the application. This is the only header file that needs to be included in the application.
`sysdep.h`	Compatibility file for 2.2, 2.4 and 2.6 kernels.
`gsc_start`	Script to detect how many 16AO12 boards are installed, and create a node for each of the cards.
`app.mak`	Makefile for the test application. Invoked as 'make –f app.mak'
`Makefile`	Makefile for the driver. Invoked as simply 'make' or 'make –f Makefile'

2.2.1. Installation on Target System

To install the driver:

1. Power down the target computer and install the 16AO12 hardware card(s).

2. Power up the system.

3. Decompress the distribution tar file in a convenient directory. To decompress the archive, use a command like:

```
tar –xzvf gsc_16aoDriver_v1.9.2.tar.gz
```

2.2.2. Building the driver and test application

To build the driver, change to the directory containing the driver source. Then:

1. Type `make` to build the driver. The driver should build without errors or warnings. If there are warnings or errors, ensure that the current kernel source is located at /usr/src/linux-2.4, or whatever settings are required for other versions of the kernel.

2. Type `make –f app.mak` to build the test application.

2.2.3. Manual Startup

Login as root. Type `./gsc_start` to start the driver. The script will display:

```
Loading module 16AO12 ...
```

2.2.4. Automatic startup

The 16AO12 driver may also be started automatically upon reboot. For auto-start using the 2.4 kernel, edit the file `/etc/rc.d/rc.local` and add the line `/<path>/gsc_start` where `<path>` is the path to where the driver and startup script are installed. Note that the startup script must be in the same directory as the driver for the startup script to work.

2.2.5. Verification of startup

There are three ways to verify that the driver has started. The first is to type:

```
ls /dev/16ao12*
```

The operating system will list an entry for each board installed, such as:

```
/dev/16AO120
```

The number appended to the name (0,1,…) is the index of the card. When multiple devices are present in the system, this is how the application specifies which device to open.

The second method is to check the /proc file system. The 16AO12 driver creates an entry in the /proc file system to report the number of boards found and the build date of the driver. Typing:

```
cat /proc/16AO12
```

Returns a result similar to:

```
version: 1.9.1

built: Jan 26 2006, 11:06:29
```

If the driver is a debug build, additional statistics are reported in the proc file.

The third method is to check the loaded modules by typing:

```
lsmod
```

The 16AO12 will be listed as one of the installed modules.

2.2.6. Verification

The distribution includes a test application and source called `testapp`. Run this application to verify that the installation was successful. The board will be opened, reset and the configuration register will be read. To run the application, type:

```
./testapp 0
```

Where 0 is the index for the first (or only) 16AO12 board installed in the system. Testapp will display a result similar to:

```
about to open /dev/gsc_16ao120
before reset: BCR is 0x10
board reset OK
```

```
after reset: BCR is 0x10
auto calibration OK
before PIO write
after PIO write
before DMA write
after DMA write
```

2.2.7. Removal

Follow the below steps to remove the driver.

1. Type:

   ```
   rmmod
   ```

2. Verify that the driver has been removed by typing:

   ```
   lsmod
   ```

   And verifying that the driver is no longer listed.

2.2.8. Uninstall

First remove the driver as described above. Remove the auto-start entry in `/etc/rc.d/rc.local`. Delete the directory containing the driver, startup script and source.

3. Sample Application

The sample application demonstrates opening the driver, and reading and writing registers using the IOCTL service. Data may be written to and read from the other registers on the hardware using similar code.

4. Driver Interface

The 16AO12 driver conforms to the device driver standards required by the LINUX driver model. The device driver provides a standard driver interface to the GSC 16AO12 board for Linux applications. The interface includes various macros, data types and functions, all of which are described in the following paragraphs. The 16AO12 specific portion of the driver interface is defined in the header file `gsc16ao12_ioctl.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.

4.1. Registers

The following table gives the complete set of 16AO12 registers. The tables are divided by register categories. All registers are accessed as 32-bits.

4.1.1. GSC Registers

The following table gives the complete set of GSC specific 16AO12 registers. For detailed definitions of these registers refer to the *16AO12 Reference Manual*. The macro fields are defined to simplify access to the registers.

Macro	Description
`BOARD_CTRL_REG`	Board Control Register (BCR)
`CHANNEL_SELECTION_REG`	Channel Select Register (CSR)
`RATE_CTRL_REG`	Sample Rate Register (SCR)
`BUFFER_OPS_REG`	Buffer Operations Register (BOR)
`RESERVED_1`	Not Used
`RESERVED_2`	Not Used
`OUTPUT_BUF_REG`	Output Buffer Register (ODB)
`ADJUSTABLE_CLOCK`	Clock Control Register (ACR) (Optional)

4.1.2. PLX PCI 9080 Registers

For detailed definitions of these registers refer to the *PCI9080 Data Book.* There is rarely any need to examine the PLX registers; they are mentioned here for completeness.

4.2. Data Types

This driver interface includes the following data types, which are defined in `gsc16ao12_ioctl.h`.

4.2.1. 16AO12_REGISTER_PARAMS

This is the data structure used to move data back and forth between the application and the 16AO12 registers.

Definition

```
typedef struct REGISTER_PARAMS
{
    ULONG eRegister;
    ULONG ulValue;
} REGISTER_PARAMS, *REGISTER_PARAMS;
```

Field	Description
`eRegister`	The register to read or write. This is the 32-bit address, not the 8-bit address shown in the

	hardware documentation. The register macros are 32-bit addresses
`ulValue`	The value to be written to the register on write, or the value read from the register on read.

4.3. read()

The 16AO12 does not support a read operation, as the board has neither data storage nor synchronous data reception capability. Reading data from the appropriate registers is done using the ioctl function.

4.4. write()

The `write()` function is used to transfer data from the user buffer to the hardware output buffer. The hardware output buffer drives the D/A output circuits.

The driver only checks one thing about the state of the output buffer: Whether the "low quarter" bit is set in Board Control Register. To avoid overrunning the buffer, the driver will only write to the buffer if the status flag indicates that the buffer level is below the threshold. If the buffer level is greater than the threshold, the driver will set and interrupt and sleep until the level drops below the threshold.

Prototype

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

Argument	Description
`fd`	This is the file descriptor of the device to access.
`buf`	Pointer to the user data buffer.
`count`	Requested number of bytes to write. This must be a multiple of four (4).

Return Value	Description
`Less than 0`	An error occurred. Consult `errno`.
`Greater than 0`	The operation succeeded. Return value is bytes written.

Example:

```
#include <errno.h>
#include <stddef.h>
#include <stdio.h>
#include <unistd.h>
#include "gsc16ao12_ioctl.h"

int 16ao12_write(int fd, __u32 *buf, size_t samples)
{
    size_t  bytes;
    int     status;

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

    if (status == -1)
        printf("write() failure, errno = %d\n", errno);
    else
        status  /= 4; // convert bytes to words

    return(status);
}
```

4.5. ioctl()

This function is the entry point to performing setup and control operations on a 16AO12 board. This function should only be called after a successful open of the device. The general form of the `ioctl` call is:

```
int ioctl(int fd, int command);
```

or

```
int ioct(int fd, int command, arg*);
```

where:

`fd`	File handle for the driver. Returned from the open() function.
`command`	The command to be performed.
`arg*`	(optional) pointer to parameters for the command. Commands that have no parameters (such as IOCTL_GSC_NO_COMMAND) will omit this parameter, and use the first form of the call.

The specific operation performed varies according to the `command` argument. The `command` argument also governs the use and interpretation of any additional arguments. The set of supported ioctl services is defined in the following sections.

Usage of all IOCTL calls is similar. Below is an example of a call using `IOCTL_READ_REGISTER` to read the contents of the board control register (BCR):

```
#include "gsc16ao12_ioctl.h"

int ReadTest(int fd)
{
    device_register_params RegPar;
    int res;

    regdata.eRegister = BOARD_CTRL_REG;
    regdata.ulValue = 0x0000; // to make sure it changes.
    res = ioctl(fd, (unsigned long)
                IOCTL_READ_REGISTER, &regdata);
    if (res < 0) {
        printf("%s: ioctl IOCTL_READ_REGISTER failed\n", argv[0]);
        }
    return (res);
```

4.5.1. IOCTL_NO_COMMAND

This is an empty driver entry point. This ioctl may be given to verify that the driver is correctly installed and that a 16AO12 device has been successfully opened. If an error status is returned verify that the driver was opened properly.

Usage

`ioctl()` **Argument**	**Description**
`Operation`	`IOCTL_NO_COMMAND`
`Arg`	None

4.5.2. IOCTL_READ_REGISTER

This operation reads the selected register and returns the value. Refer to gsc`16ao12_ioctl.h` for a complete list of the accessible registers.

Usage

`ioctl()` **Argument**	**Description**
`Operation`	`IOCTL_READ_REGISTER`
`Arg`	`REGISTER_PARAMS*`

4.5.3. IOCTL_WRITE_REGISTER

This service writes a value to a 16AO12 register. The accessible registers are listed in `gsc16ao12_ioctl.h`.

Usage

`ioctl()` **Argument**	**Description**
`request`	`IOCTL_WRITE_REGISTER`
`Arg`	`REGISTER_PARAMS*`

4.5.4. IOCTL_INIT_BOARD

This service initializes the hardware to a known state.

Usage

`ioctl()` **Argument**	**Description**
`request`	`IOCTL_INIT_BOARD`
`Arg`	`None`

4.5.5. IOCTL_SET_DMA_MODE

This service selects the mode for writing data to the output buffer, either DMA or PIO (Programmed I/O). Possible choices are:

```
DMA_DISABLE
DMA_ENABLE
```

Usage

`ioctl()` **Argument**	**Description**
`Request`	`IOCTL_SET_DMA_MODE`
`Arg`	`Unsigned long*`

4.5.6. IOCTL_GET_DEVICE_ERROR

This service returns the most recent error detected by the hardware. Possible return values are:

```
DEVICE_SUCCESS
DEVICE_INVALID_PARAMETER
DEVICE_INVALID_BUFFER_SIZE
DEVICE_PIO_TIMEOUT
DEVICE_DMA_TIMEOUT
DEVICE_IOCTL_TIMEOUT
DEVICE_OPERATION_CANCELLED
DEVICE_RESOURCE_ALLOCATION_ERROR
DEVICE_INVALID_REQUEST
DEVICE_AUTOCAL_FAILED
```

Usage

`ioctl()` Argument	**Description**
`request`	`IOCTL_GET_DEVICE_ERROR`
`Arg`	`Unsigned long*`

4.5.7. IOCTL_AUTOCALIBRATE

This service performs an autocalibration service on the hardware. The IOCTL waits for the "done" interrupt before returning.

Usage

`ioctl()` Argument	**Description**
`request`	`IOCTL_AUTOCALIBRATE`
`Arg`	`None`

4.5.8. IOCTL_PROGRAM_RATE_GEN

This service writes the passed value to the rate generator register. See the hardware manual for a description of how to select a value for the rate generator.

Usage

`ioctl()` Argument	**Description**
`request`	`IOCTL_PROGRAM_RATE_GEN`
`Arg`	`Unsigned long*`

4.5.9. IOCTL_CHANNEL_ENABLE

This service selects what channels will be written during a write cycle. Individual channel masks should be OR-ed together. Possible values are:

```
CHAN_0_ENABLE
CHAN_1_ENABLE
```

```
CHAN_2_ENABLE
CHAN_3_ENABLE
CHAN_4_ENABLE
CHAN_5_ENABLE
CHAN_6_ENABLE
CHAN_7_ENABLE
CHAN_8_ENABLE
CHAN_9_ENABLE
CHAN_10_ENABLE
CHAN_11_ENABLE
CHAN_ALL_ENABLE
```

Usage

`ioctl()` Argument	**Description**
`Request`	`IOCTL_CHANNEL_ENABLE`
`Arg`	`Unsigned long*`

4.5.10. IOCTL_SET_OUT_BUFFER_SIZE

This service sets the size of the virtual output buffer. Possible values are:

```
OUT_BUFFER_SIZE_4
OUT_BUFFER_SIZE_8
OUT_BUFFER_SIZE_16
OUT_BUFFER_SIZE_32
OUT_BUFFER_SIZE_64
OUT_BUFFER_SIZE_128
OUT_BUFFER_SIZE_256
OUT_BUFFER_SIZE_512
OUT_BUFFER_SIZE_1024
OUT_BUFFER_SIZE_2048
OUT_BUFFER_SIZE_4096
OUT_BUFFER_SIZE_8192
OUT_BUFFER_SIZE_16384
OUT_BUFFER_SIZE_32768
OUT_BUFFER_SIZE_65536
```

Usage

`ioctl()` Argument	**Description**
`Request`	`IOCTL_SET_OUT_BUFFER_SIZE`
`Arg`	`Unsigned long*`

4.5.11. IOCTL_GET_BUFFER_STATUS

This service reads the status of the output buffer register. Possible return values are:

```
OUTPUT_EMPTY
OUTPUT_LOW_QTR
OUTPUT_HIGH_QTR
```

Usage

`ioctl()` **Argument**	**Description**
`Request`	`IOCTL_GET_BUFFER_STATUS`
`Arg`	`Unsigned long*`

4.5.12. IOCTL_ENABLE_CLK

This service enables the output clock.

Usage

`ioctl()` **Argument**	**Description**
`request`	`IOCTL_ENABLE_CLK`
`Arg`	`None`

4.5.13. IOCTL_DISABLE_CLK

This service disables the output clock.

Usage

`ioctl()` **Argument**	**Description**
`request`	`IOCTL_DISABLE_CLK`
`Arg`	`None`

4.5.14. IOCTL_SELECT_DATA_FORMAT

This service selects the format of the data written to the output buffer. Possible values are:

```
TWOS_COMP
OFFSET_BINARY
```

Usage

`ioctl()` **Argument**	**Description**
`Request`	`IOCTL_SELET_DATA_FORMAT`
`Arg`	`Unsigned long*`

4.5.15. IOCTL_SELECT_SAMPLING_MODE

This service selects whether the hardware will sample in continuous mode or burst mode. Possible values are:

```
CONT_MODE
BURST_MODE
```

Usage

`ioctl()` **Argument**	**Description**
`Request`	`IOCTL_SELECT_SAMPLING_MODE`
`Arg`	`Unsigned long*`

4.5.16. IOCTL_GET_BURSTING_STATUS

This service returns the burst status of the output buffer. Possible return values are:

```
BURST_NOT_READY
BURST_READY
```

Usage

`ioctl()` **Argument**	**Description**
`Request`	`IOCTL_GET_BURSTING_STATUS`
`Arg`	`Unsigned long*`

4.5.17. IOCTL_BURST_TRIGGER

This service starts a burst-mode transfer of data from the output buffer to the active output channels.

Usage

`ioctl()` **Argument**	**Description**
`request`	`IOCTL_BURST_TRIGGER`
`Arg`	`none`

4.5.18. IOCTL_ENABLE_REMOTE_GND_SENSE

This service sets the hardware to use the remote ground sense feature.

Usage

`ioctl()` **Argument**	**Description**
`request`	`IOCTL_ENABLE_LOCAL_GND_SENSE`
`Arg`	`None`

4.5.19. IOCTL_DISABLE_LOCAL_GND_SENSE

This service disables the hardware remote ground sense and uses the internal ground sense.

Usage

`ioctl()` **Argument**	**Description**
`request`	`IOCTL_DISABLE_LOCAL_GND_SENSE`
`Arg`	`None`

4.5.20. IOCTL_SELECT_OUT_CLKING_MODE

This service selects the mode for the output clock. Possible values are:

```
SEQUENTIAL
SIMULTANEOUS
```

Usage

`ioctl()` Argument	Description
`Request`	`IOCTL_SELECT_OUT_CLKING_MODE`
`Arg`	`Unsigned long*`

4.5.21. IOCTL_SELECT_CLK_SOURCE

This service selects the source for the output clock. Possible values are:

```
INTERNAL
EXTERNAL
```

Usage

`ioctl()` Argument	Description
`Request`	`IOCTL_SELECT_CLK_SOURCE`
`Arg`	`Unsigned long*`

4.5.22. IOCTL_GET_CLK_STATUS

This service returns the current clock status. Possible values are:

```
CLOCK_NOT_READY
CLOCK_READY BOR_CLOCK_READY
```

Usage

`ioctl()` Argument	Description
`request`	`IOCTL_GET_CLK_STATUS`
`Arg`	`Unsigned long*`

4.5.23. IOCTL_SINGLE_OUTPUT_CLK_EVENT

This service generates a clock strobe.

Usage

`ioctl()` Argument	Description
`request`	`IOCTL_SINGLE_OUTPUT_CLK_EVENT`
`Arg`	`None`

4.5.24. IOCTL_SELECT_BUF_CONFIG

This service selects whether the output buffer is closed (circular) or open. Possible values are:

```
OPEN_BUF
CIRCULAR_BUF
```

Usage

`ioctl()` Argument	**Description**
`request`	`IOCTL_SELECT_BUF_CONFIG`
`Arg`	`Unsigned long*`

4.5.25. IOCTL_LOAD_ACCESS_REQ

This service requests access to the circular buffer when it is closed.

Usage

`ioctl()` Argument	**Description**
`request`	`IOCTL_LOAD_ACCESS_REQ`
`Arg`	`None`

4.5.26. IOCTL_GET_CIR_BUF_STATUS

This service requests the current status of the circular buffer. Possible return values are:

```
CIR_BUF_NOT_READY
CIR_BUF_READY
```

Usage

`ioctl()` Argument	**Description**
`request`	`IOCTL_GET_CIR_BUF_STATUS`
`Arg`	`Unsigned long*`

4.5.27. IOCTL_CLEAR_BUFFER

This service empties all residual data from the output buffer.

Usage

`ioctl()` Argument	**Description**
`request`	`IOCTL_CLEAR_BUFFER`
`Arg`	`None`

4.5.28. IOCTL_GET_DEVICE_TYPE

This service returns an enumeration indicating what member of the 16AO family is attached. Possible return values are:

```
GSC_16AO_12
GSC_16AO_2
```

Usage

`ioctl()` Argument	Description
`Request`	`IOCTL_GET_DEVICE_TYPE`
`Arg`	`Unsigned long*`

4.5.29. IOCTL_SET_TIMEOUT

This service sets the timeout for initializing, auto-calibration and write operations. Value passed is in seconds.

Usage

`ioctl()` Argument	Description
`Request`	`IOCTL_SET_TIMEOUT`
`Arg`	`Unsigned long*`

4.5.30. IOCTL_SET_BIG_ENDIAN

Tells the driver that the system is big endian or little endian. Default is little endian. Currently setting to big endian only causes byte swapping during DMA. Possible values are:

TRUE: Use big endian swapping.
FALSE: Use little endian mode (no swapping, default).

Usage

`ioctl()` Argument	Description
`Request`	`IOCTL_SET_TIMEOUT`
`Arg`	`Unsigned long*`

Document History

Revision	**Description**
December 1, 2004	Initial release.
March 17, 2005	Corrected description of 'write' operation.
February 16, 2006	Removed GET_DEVICE_TYPE IOCTL. Corrected IOCTL header file name.
February 24, 2006	Corrected some typos.