12AI64/16AI64

64 Channel, 12 or 16-Bit Analog Input Board

PMC-12AI64
PMC-16AI64
PMC-16AI64-SS
PCI-16AI64-SS
PCI-16AI64-SSA

LINUX Device Driver User Manual

Manual Revision: August 30, 2005

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.**

Table of Contents

1. Introduction

1.1. Purpose

The purpose of this document is to describe the interface to the 16AI64 LINUX device driver. This software provides the interface between "Application Software" and the 16AI64 family of boards, including the 12AI64, 16AI64, 16AI64SS and 16AI64SSA. The label 16AI64 will be used generically throughout this document to refer to any member of the board family.

1.2. Acronyms

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

Acronyms	Description
DMA	Direct Memory Access
PIO	Programmed I/O
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
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 16AI64 driver software executes under control of the Linux operating system. The device driver allows the user to open and close one or more 16AI64 boards, then read and write data to the registers on the hardware. Note that different board models may be used on the same system.

1.5. Hardware Overviews

1.5.1. PMC-12AI64

The high-density PMC-12AI64 board provides cost effective 1,500,000 conversions-per-second 12-bit analog input capabilities in a single width PMC format. The inputs are configurable either as 64 single-ended channels or as 32 differential channels, and the input range can be software selectable as ±10V, ±5V or ±2.5V. Scan rates can be controlled from either (a) an internal rate generator, (b) through an external digital input, or (c) by direct software commands. Multiple PMC-12AI64 boards can be connected together for synchronous scanning. Data buffering is accomplished through a 64K sample FIFO. Internal auto calibration networks permit calibration to be performed without removing the board from the system.

1.5.2. PMC-16AI64

The high-density PMC-16AI64 board provides a cost effective 500,000 conversions-per-second 16-bit analog input capability in a single-width PMC format. The inputs are configurable either as 64 single-ended channels or as 32 differential channels, and the input range can be software selectable as ±10V, ±5V or ±2.5V. Scan rates can be controlled from either (a) an internal rate generator, (b) through an external digital input, or (c) by direct software commands. Multiple PMC-16AI64 boards can be connected together for synchronous scanning. A 64K-sample

FIFO provides maximum buffering of input data. Internal autocalibration networks permit calibration to be performed without removing the board from the system.

1.5.3. PMC-16AI64SS, PCI-16AI64SS

The 16-Bit PMC-16AI64SS and PCI-16AI64SS (**S**ynchronous **S**ampling) analog input boards sample and digitizes 64 input channels simultaneously at rates up to 200,000 samples per second for each channel. The resulting 16-bit sampled data is available to the PCI bus through a 64K-Sample FIFO buffer. Each input channel contains a dedicated 16-Bit sampling ADC. All operational parameters are software configurable. Inputs can be sampled in groups of 2, 4, 8, 16, 32 or 64 channels; or any single channel can be sampled continuously. The sample clock can be generated from an internal rate generator, or by software or external hardware. Input ranges are software-selectable as ±10V, ±5V or ±2.5V. The inputs can be clocked either continuously or in triggered bursts. An on-demand auto-calibration feature determines offset and gain correction values for each input channel.

1.6. Reference Material

The following reference material may be of particular benefit in using the 16AI64 and this driver. The specifications provide the information necessary for an in depth understanding of the specialized features implemented on this board.

- The applicable *AI64 User Manual* from General Standards Corporation. Manuals are available in PDF format at the company website, *http://www.generalstandards.com*

- 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 only been tested on a PC system with dual Intel x86 processors running RedHat 7.3 with kernel 2.4.18-14SMP, and Fedora Core with kernel 2.6.8-1.521smp. The driver will have to be rebuilt for the version of the kernel running on the target machine. Support is built into the driver for version 2.2 of the kernel, but that configuration has not been tested.

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. Not required to include in an application.
`gsc16AI64_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`	A script to detect how many 16AI64 boards are installed, and create a node for each of the boards.
`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'
`.o,.ko`	Object files produced by the make process. May not be loadable unless rebuilt for the version of the kernel running on the target system. The .ko suffix is used for kernel objects by the 2.6 kernel.

2.2.1. Installation on Target System

To install the driver:

1. Power down the target computer and install the AI64 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_16AI64Driver.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 clean` to remove old object files. 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. The `Makefile` contains defines that point to the current kernel source, and modifying Makefile to point to the correct kernel source may allow the driver to build. It is also possible that a newer compiler with more stringent syntax checking

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 gsc16ai64 ...
```

This is all that will be displayed if there are no problems with the startup.

2.2.4. Automatic startup

The 16AI64 driver may also be started automatically upon reboot. For auto-start, 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/gsc16ai64*
```

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

```
/dev/gsc16ai640
```

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 16AI64 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/gsc16ai64
```

Returns a result similar to:

```
version: 2.0

built: Aug 26 2003, 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 gsc16ai64 will be listed as one of the installed modules.

Note that all board varieties are reported as gsc16ai64.

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) 16AI64 board installed in the system. Testapp will display a result similar to:

```
***** Build Jan 21 2004, 11:40:15 *****

about to open

open OK - board type: PMC-12AI64 channels: 64

before init: BCR is 0x4071

board reset OK

after init: BCR is 0x4060

autocalibration...

auto calibration OK


Total Buffer read: 1024

00622592 00622592 00622592 00622592 00622592 00622592 00622592 00622592

00622592 00622592 00622592 00622592 00622592 00622592 00622592 00622592

00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

The test program will run through 1024 read cycles then terminate.

2.2.7. Removal

Follow the below steps to remove the driver.

1. Type:

   ```
   Rmmod gsc16ai64
   ```

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.

The program loops reading of data and displaying a line of statistics. Typing CTRL-C will terminate the program. Otherwise, The program runs for 1024 read cycles then exits normally. After reading the data, the program displays the number of samples read from each channel.

4. Driver Interface

The 16AI64 driver conforms to the device driver standards required by the LINUX driver model. The device driver provides a standard driver interface to the GSC 16AI64 family of boards for Linux applications. The interface includes various macros, data types and functions, all of which are described in the following paragraphs. The 16AI64 specific portion of the driver interface is defined in the header file `gsc16AI64_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 16AI64 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 16AI64 registers. For detailed definitions of these registers refer to the *16AI64 User Manual*. The macro fields are defined to simplify access to the registers.

Macro	Description
`BOARD_CTRL_REG`	Board Control Register (BCR)
`INT_CTRL_REG`	Interrupt Control Register (ICR)
`INPUT_DATA_BUFF_REG`	Input Data Buffer Register (IDR)
`INPUT_BUFF_CTRL_REG`	Input Buffer Control Register (IDC)
`RATE_A_GENERATOR_REG`	Rate Generator "A" Register (RGRA)
`RATE_B_GENERATOR_REG`	Rate Generator "B" Register (RGRB)
`BUFFER_SIZE_REG`	Buffer Size Register
`RESERVED_1_REG`	Reserved
`SCAN_SYNCH_CTRL_REG`	Scan and Synch Control (SSCR)
`RESERVED_2_REG`	Reserved
`FIRMWARE_REV_REG`	Firmware Revision Register (FRR)*
`AUTOCAL_VALUES_REG`	Auto Calibration Results Register (ACRR)*

**SS boards only.*

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. The driver handles management of the PLX DMA and interrupt registers.

4.2. Data Types

This driver interface includes the following data types, which are defined in `gsc16AI64_ioctl.h`.

4.2.1. 16AI64_REGISTER_PARAMS

This is the data structure used to move data back and forth between the application and the 16AI64 registers.

Definition

```
typedef struct device_register_params {
     unsigned int DeviceRegister;
  unsigned long ulValue;
}DEVICE_REGISTER_PARAMS, *PDEVICE_REGISTER_PARAMS;
```

Field	**Description**
`DeviceRegister`	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 `read()` function is used to retrieve data from the driver. The application passes down the handle of the driver instance, a pointer to a buffer and the size of the buffer. The size field portion of the request is passed to the `read()` function as a number of bytes, and the number of bytes read is returned by the function.

Depending on how much data is available and what the read mode is, you may receive back less data than requested. The Linux standards only require that at least one byte be returned for a read to be successful.

How the buffer is filled is dependant on what DMA setting is active:

- **No DMA**: This is called programmed I/O or PIO. The driver will read data from the data register until either the buffer is full, or there is no more data in the input buffer, whichever comes first.

- **Regular DMA**: For a regular DMA transaction, the driver needs to determine how much data to transfer. The driver is set up to only do a DMA operation when the input buffer contains at least BUFFER_THRESHOLD samples in the buffer. So if the flags indicate that there is greater than BUFFER_THRESHOLD samples available, the driver immediately initiates a DMA transfer between the hardware and a system buffer. The driver sets an interrupt and sleeps until the DMA finished interrupt is received, then copies the data into the user buffer and returns.

  If the flags indicate that there is not enough data in the buffer, the driver sets up for an interrupt when the BUFFER_THRESHOLD is reached and sleeps. When the interrupt is received, the driver then sets up a DMA transfer as described above.

- **Demand mode DMA**: The byte count passed in the `read()`is converted to words and written to the DMA hardware. The driver sets an interrupt for DMA finished and goes to sleep. The DMA hardware transfers the requested number of words into the system (intermediate) buffer and generates an interrupt.

  The difference between regular and demand mode has to do with when the transaction is started. A demand mode transaction may be initiated at any buffer data level. The regular DMA transaction is only started when there is sufficient data.

Note that due to limitations of the Linux operating system, the driver cannot copy directly from the hardware to the user buffer. Instead, the data must pass through an intermediate DMA-capable buffer. The size of the intermediate buffer is determined by the #define DMA_ORDER in the gsc16AI64.h file. The driver attempts to allocate 2^DMA_ORDER pages. On larger systems, this number can be increased, reducing the number of operations required to transfer the data. Demand mode DMA transfers are also limited to the capacity of the intermediate buffer.

DMA always uses an intermediate system buffer then copies the resulting data into the user buffer. It is not currently possible with (version 2.4) Linux to DMA directly into a user buffer.

Example

```
#include "gsc16AI64_ioctl.h"

int res;
    res = read(fd, buffer, BUFFER_SIZE);
        if (res <= 0){
            printf("read error...res= %d\n",res);
            return (1);
        }
```

4.4. write()

The 16AI64 does not support a write operation, as the board has neither data storage nor synchronous data transmission capability. Writing data to the appropriate registers is done using the `ioctl` function.

4.5. ioctl()

This function is the entry point to performing setup and control operations on a 16AI64 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_DEVICE_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 a following section.

The 16AI64 driver implements the following ioctl services. Each service is described along with the applicable `ioctl` function arguments.

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

```
#include "gsc16AI64_ioctl.h"

int ReadTest(int fd)
```

```
{
    device_register_params RegPar;
    DWORD dwTransferSize;

    regdata.ulRegister = BOARD_CTRL_REG;
    regdata.ulValue = 0x0000; // to make sure it changes.
    res = ioctl(fd, (unsigned long)
                    IOCTL_DEVICE_READ_REGISTER, &regdata);
    if (res < 0) {
        printf("%s: ioctl IOCTL_READ_REGISTER failed\n", argv[0]);
        return (1);
    }
```

The following sections describe the individual IOCTLs and list the parameters for each.

4.5.1. IOCTL_DEVICE_NO_COMMAND

`IOCTL_DEVICE_NO_COMMAND` is an empty driver entry point. This ioctl may be given to verify that the driver is correctly installed and that a 16AI64 device has been successfully opened. If an error status is returned verify that the driver was opened properly.

Usage

`ioctl()` Argument	**Description**
`Operation`	`IOCTL_DEVICE_NO_COMMAND`
`arg`	None

4.5.2. IOCTL_DEVICE_INIT_BOARD

`IOCTL_DEVICE_INIT_BOARD` Perform an initialization cycle on the board. The driver writes to the BCR to initiate an initialization cycle, and then waits for an interrupt indicating that the cycle completed before returning from the call.

Usage

`ioctl()` Argument	**Description**
`Operation`	`IOCTL_DEVICE_INIT_BOARD`
`arg`	None

4.5.3. IOCTL_DEVICE_GET_DEVICE_TYPE

`IOCTL_DEVICE_GET_DEVICE_TYPE` returns an enumeration of the type of AI64 board the driver has opened. Possible return values are:

`gsc12ai64`	`A GSC 12AI64 board`
`gsc16ai64`	`A GSC 16AI64 board`
`gsc16ai64ss`	`A GSC 16AI64SS board`

Usage

`ioctl()` **Argument**	**Description**
`Operation`	`IOCTL_DEVICE_GET_DEVICE_TYPE`
`Arg`	`Unsigned long *pType`

4.5.4. IOCTL_DEVICE_READ_REGISTER

`IOCTL_DEVICE_READ_REGISTER` reads the selected register and returns the value. Refer to `gsc16AI64_ioctl.h` for a complete list of the accessible registers. Note that all board functions may be controlled using IOCTL_DEVICE_READ_REGISTER and IOCTL_DEVICE_WRITE_REGISTER. Other IOCTLs are provided for ease of use.

Usage

`ioctl()` **Argument**	**Description**
`Operation`	`IOCTL_DEVICE_READ_REGISTER`
`Arg`	`DEVICE_REGISTER_PARAMS*`

4.5.5. IOCTL_DEVICE_WRITE_REGISTER

`IOCTL_DEVICE_WRITE_REGISTER` writes the passed value to the selected register. Refer to `gsc16AI64_ioctl.h` for a complete list of the accessible registers. Note that all board functions may be controlled using IOCTL_DEVICE_READ_REGISTER and IOCTL_DEVICE_WRITE_REGISTER. Other IOCTLs are provided for ease of use.

Usage

`ioctl()` **Argument**	**Description**
`Operation`	`IOCTL_DEVICE_READ_REGISTER`
`Arg`	`DEVICE_REGISTER_PARAMS*`

4.5.6. IOCTL_DEVICE_ENABLE_DMA

`16AI64_IOCTL_ENABLE_DMA` enables or disables DMA data transfer from the hardware to the system data buffer. It also selects which DMA method to use.

Possible arg values are:

DMA_DISABLE	(default) Set to use PIO data transfer. The driver will copy data from the hardware until either the FIFO is empty, or the requested number of samples has been copied.
DMA_ENABLE	Set to enable standard DMA. The driver will DMA data from the hardware when the quantity of data in the input buffer reaches the value in the threshold register.
DMA_DEMAND_MODE	Use demand-mode DMA. The DMA hardware will transfer data from the input buffer as it becomes available. DMA will terminate when the requested number of bytes have been transferred*.

* NOTE – Demand mode is only available on the SS models.

Usage

`ioctl()` Argument	Description
`command`	`16AI64_IOCTL_ENABLE_DMA`
`arg`	`unsigned long *pDMAMode`

4.5.7. IOCTL_DEVICE_GET_DEVICE_ERROR

`IOCTL_DEVICE_GET_DEVICE_ERROR` returns the most recent error detected by the driver. The result is returned in the argument whose pointer is passed by the application. Possible return values are:

`DEVICE_SUCCESS`	No error detected.
`DEVICE_INVALID_PARAMETER`	The parameter passed to the ioctl was invalid.
`DEVICE_INVALID_BUFFER_SIZE`	The passed buffer size is greater than the device buffer size.
`DEVICE_PIO_TIMEOUT`	An attempted PIO data read timed out.
`DEVICE_DMA_TIMEOUT`	An attempted DMA data read timed out.
`DEVICE_IOCTL_TIMEOUT`	A general ioctl operation timed out.
`DEVICE_OPERATION_CANCELLED`	A wait state was cancelled by the system.
`DEVICE_RESOURCE_ALLOCATION_ERROR`	The driver was unable to allocate required resources.
`DEVICE_INVALID_REQUEST`	An invalid ioctl was passed to the driver.
`DEVICE_AUTOCAL_FAILED`	A requested auto calibration of the analog section failed.

Usage

`ioctl()` Argument	Description
`command`	`IOCTL_DEVICE_GET_DEVICE_ERROR`
`arg`	`unsigned long *pError`

4.5.8. IOCTL_DEVICE_INPUT_RANGE_CONFIG

`IOCTL_DEVICE_INPUT_RANGE_CONFIG` selects the input range of the analog input channels. Possible arg values are:

`INPUT_RANGE_2_5`	2.5 volts full scale
`INPUT_RANGE_5`	5 volts full scale
`INPUT_RANGE_10`	10 volts full scale (default)

Usage

`ioctl()` Argument	Description
`command`	`IOCTL_DEVICE_INPUT_RANGE_CONFIG`
`arg`	`unsigned long *pInputRange`

4.5.9. IOCTL_DEVICE_INPUT_MODE_CONFIG

`IOCTL_DEVICE_INPUT_MODE_CONFIG` is used to set the input configuration. For non-Synchronous Sampling (SS) boards, the default is differential. The SS boards default to "system analog input," which is single-ended. See `IOCTL_DEVICE_SS_MODE` for more options specific to the SS boards.

Possible arguments are:

`INPUT_DIFFERENTIAL`	Differential input (default for non-SS boards)
`INPUT_SINGLE_ENDED`	Single-ended input
`INPUT_ZERO_TEST`	All inputs internally tied to ground
`INPUT_VREF_TEST`	All inputs internally tied to Vref
`INPUT_SYSTEM_ANALOG_SS`	System analog input (Default for the SS boards)

Usage

`ioctl()` Argument	**Description**
`Command`	`IOCTL_DEVICE_INPUT_MODE_CONFIG`
`Arg`	`unsigned long *iMode`

4.5.10. IOCTL_DEVICE_SS_MODE_CONFIG

`IOCTL_DEVICE_SS_MODE_CONFIG` sets Synchronous Sampling (SS) differential processing features. The differential processing features are specific to the SS boards.

Possible values for arg are:

`SS_SINGLE_ENDED`	Processing of input data is limited to gain and offset error processing (default)
`SS_PSEUDO_DIFFERENTIAL`	Channel 00 is the input LO reference for all of the other channels
`SS_FULL_DIFFERENTIAL`	Each odd-numbered channel is the LO reference for each even-numbered HI channel.

Usage

`ioctl()` Argument	**Description**
`command`	`IOCTL_DEVICE_SS_MODE_CONFIG`
`arg`	`unsigned long *iMode`

4.5.11. IOCTL_DEVICE_INPUT_FORMAT_CONFIG

`IOCTL_DEVICE_INPUT_FORMAT_CONFIG` selects the output data format. Possible values for arg are:

`FORMAT_TWO_COMPLEMENT`	Format the output data as two's complement
`FORMAT_OFFSET_BINARY`	Format the output data as offset binary

Usage

`ioctl()` Argument	**Description**
`command`	`IOCTL_DEVICE_INPUT_FORMAT_CONFIG`
`arg`	`unsigned long *pFormat`

4.5.12. IOCTL_DEVICE_START_AUTOCAL

`IOCTL_DEVICE_START_AUTOCAL` initiates an auto calibration cycle. The driver will wait for the end of cycle interrupt before returning from the call. There are no arguments for this call.

Usage

`ioctl()` **Argument**	**Description**
`command`	`IOCTL_DEVICE_START_AUTOCAL`
`arg`	None

4.5.13. IOCTL_DEVICE_START_SINGLE_SCAN

`IOCTL_DEVICE_START_SINGLE_SCAN` initiates a single scan cycle. There are no arguments for this call.

Usage

`ioctl()` **Argument**	**Description**
`command`	`IOCTL_DEVICE_START_SINGLE_SCAN`
`arg`	None

4.5.14. IOCTL_DEVICE_SET_SCAN_SIZE

`IOCTL_DEVICE_SET_SCAN_SIZE` selects the number of channels to scan with each pass. Possible values for arg are:

`SCAN_SINGLE_CHAN`	Single channel only
`SCAN_2_CHAN`	Scan two channels
`SCAN_4_CHAN`	Scan four channels
`SCAN_8_CHAN`	Scan eight channels
`SCAN_16_CHAN`	Scan sixteen channels
`SCAN_32_CHAN`	Scan 32 channels
`SCAN_64_CHAN`	Scan 64 channels

Usage

`ioctl()` **Argument**	**Description**
`command`	`16AI64_IOCTL_ENABLE_DMA`
`arg`	`unsigned long *pScanSize`

4.5.15. IOCTL_DEVICE_SET_SCAN_CLOCK

`IOCTL_DEVICE_SET_SCAN_CLOCK` is used to select the source for the scan clock. Possible values for arg are:

`INTERNAL_RATE_A`	Use internal rate generator "A" for clocking.
`INTERNAL_RATE_B`	Use internal rate generator "B" for clocking.
`EXTERNAL_SYNCH`	Use the external synch input for clocking
`BCR_INPUT_SYNCH`	Clocking on Board Control Register (BCR) Input Synch bit set

Usage

`ioctl()` **Argument**	**Description**
`Command`	`16AI64_IOCTL_ENABLE_DMA`
`Arg`	`unsigned long *pSource`

4.5.16. IOCTL_DEVICE_DISABLE_PCI_INTERRUPTS

`IOCTL_DEVICE_DISABLE_PCI_INTERRUPTS` disables PCI interrupts from the hardware. Not required for normal operation, may be used at program termination to ensure that no spurious interrupts are generated.

Usage

`ioctl()` Argument	**Description**
`Command`	`IOCTL_DEVICE_DISABLE_PCI_INTERRUPTS`
`Arg`	`None`

4.5.17. IOCTL_DEVICE_SET_B_CLOCK_SOURCE

`IOCTL_DEVICE_SET_B_CLOCK_SOURCE` selects the source for the "B" clock. Possible values are:

`MASTER_CLOCK`	The master clock is fed directly into the "B" clock
`RATE_A_OUTPUT`	The output of the "A" clock is fed into the "B" clock

Usage

`ioctl()` Argument	**Description**
`command`	`IOCTL_DEVICE_SET_B_CLOCK_SOURCE`
`arg`	`unsigned long * pSource`

4.5.18. IOCTL_DEVICE_SINGLE_CHAN_SELECT

`IOCTL_DEVICE_SINGLE_CHAN_SELECT` selects which channel to use for a single-channel scans. Possible values are `0-63.`

Usage

`ioctl()` Argument	**Description**
`command`	`IOCTL_DEVICE_SINGLE_CHAN_SELECT`
`arg`	`unsigned long * channel`

4.5.19. IOCTL_DEVICE_SET_SCAN_RATE_A

`IOCTL_DEVICE_SET_SCAN_RATE_A` sets the scan rate divisor for scan clock "A." Possible values are `0-0xFFFF.`

Usage

`ioctl()` Argument	**Description**
`Command`	`IOCTL_DEVICE_SET_SCAN_RATE_A`
`Arg`	`unsigned long *pSetting`

4.5.20. IOCTL_DEVICE_SET_SCAN_RATE_B

IOCTL_DEVICE_SET_SCAN_RATE_B sets the scan rate divisor for scan clock "B." Possible values are `0-0xFFFF.`

Usage

`ioctl()` Argument	**Description**
`Command`	`IOCTL_DEVICE_SET_SCAN_RATE_B`
`Arg`	`unsigned long *pSetting`

4.5.21. IOCTL_DEVICE_GEN_DISABLE_A

`IOCTL_DEVICE_GEN_DISABLE_A` disables rate generator "A."

Usage

`ioctl()` Argument	**Description**
`Command`	`IOCTL_DEVICE_GEN_DISABLE_A`
`Arg`	None

4.5.22. IOCTL_DEVICE_GEN_DISABLE_B

`IOCTL_DEVICE_GEN_DISABLE_B` disables rate generator "B."
Usage

`ioctl()` Argument	**Description**
`Command`	`IOCTL_DEVICE_GEN_DISABLE_B`
`Arg`	None

4.5.23. IOCTL_DEVICE_GEN_ENABLE_A

`IOCTL_DEVICE_GEN_ENABLE_A` enables rate generator "A."

Usage

`ioctl()` Argument	**Description**
`Command`	`IOCTL_DEVICE_GEN_ENABLE_A`
`Arg`	None

4.5.24. IOCTL_DEVICE_GEN_ENABLE_B

`IOCTL_DEVICE_GEN_ENABLE_B` enables rate generator "B."
Usage

`ioctl()` Argument	**Description**
`Command`	`IOCTL_DEVICE_GEN_ENABLE_B`
`Arg`	None

4.5.25. IOCTL_DEVICE_SET_BUFFER_THRESHOLD

`IOCTL_DEVICE_SET_BUFFER_THRESHOLD` writes the passed value to the buffer threshold register. The buffer threshold is used to generate interrupts telling the device driver when there is a known quantity of data in the input buffer. Range = `0-0xFFFF`

Usage

`ioctl()` **Argument**	**Description**
`Command`	`IOCTL_DEVICE_SET_BUFFER_THRESHOLD`
`Arg`	`unsigned long *pThreshold`

4.5.26. IOCTL_DEVICE_CLEAR_BUFFER

`IOCTL_DEVICE_CLEAR_BUFFER` clears all data from the input buffer.

Usage

`ioctl()` **Argument**	**Description**
`command`	`IOCTL_DEVICE_CLEAR_BUFFER`
`arg`	None

4.5.27. IOCTL_DEVICE_DISABLE_BUFFER_SS

`IOCTL_DEVICE_DISABLE_BUFFER_SS` disables the input buffer. This command is only valid for the SS boards.

Usage

`ioctl()` **Argument**	**Description**
`Command`	`IOCTL_DEVICE_DISABLE_BUFFER`
`Arg`	None

4.5.28. IOCTL_DEVICE_ENABLE_BUFFER_SS

`IOCTL_DEVICE_ENABLE_BUFFER_SS` enables the input buffer. This is the default state. This command is only valid for the SS boards.

Usage

`ioctl()` **Argument**	**Description**
`Command`	`IOCTL_DEVICE_DISABLE_BUFFER`
`Arg`	None

4.5.29. IOCTL_DEVICE_DISABLE_DMA_ALMOST_SS

`IOCTL_DEVICE_DISABLE_DMA_ALMOST_SS` sets the demand-mode DMA to terminate when the buffer becomes empty. This is the default state. This command is only valid for the SS boards.

Usage

`ioctl()` **Argument**	**Description**
`Command`	`IOCTL_DEVICE_DISABLE_DMA_ALMOST_SS`
`Arg`	None

4.5.30. IOCTL_DEVICE_ENABLE_DMA_ALMOST_SS

`IOCTL_DEVICE_ENABLE_DMA_ALMOST_SS` sets the demand-mode DMA to terminate when the buffer becomes empty. This is the default state. This command is only valid for the SS boards.

Usage

`ioctl()` Argument	**Description**
`command`	`IOCTL_DEVICE_ENABLE_DMA_ALMOST_SS`
`arg`	None

4.5.31. IOCTL_DEVICE_SET_TIMEOUT

`IOCTL_DEVICE_SET_TIMEOUT` sets the timeout (in seconds) for board operations. The default is five seconds.

Usage

`ioctl()` Argument	**Description**
`command`	`IOCTL_DEVICE_SET_TIMEOUT`
`arg`	`unsigned long *pTimeout`

4.5.32. IOCTL_DEVICE_FILL_BUFFER

`IOCTL_DEVICE_FILL_BUFFER` instructs the driver to completely fill the user buffer before returning when set to `TRUE`. Normally, By Linux convention, a `read()` operation is only required to return one or more data samples. When this value is set to `FALSE`, the driver will only return at most the capacity of the intermediate buffer. Default is `FALSE`.

Note that due to limitations of the Linux operating system, the driver cannot copy directly from the hardware to the user buffer. Instead, the data must pass through an intermediate DMA-capable buffer. The size of the intermediate buffer is determined by the #define DMA_ORDER in the gsc16AI64.h file. The driver attempts to allocate 2^DMA_ORDER pages. On larger systems, this number can be increased, reducing the number of operations required to transfer the data.

Usage

`ioctl()` Argument	**Description**
`Command`	`IOCTL_DEVICE_SET_TIMEOUT`
`Arg`	`unsigned long *pSetting`

4.5.33. IOCTL_DEVICE_DETECT_OVERFLOW

`IOCTL_DEVICE_DETECT_OVERFLOW` is used to instruct the driver to fail a read() operation if the input buffer becomes full. Because there is no direct way to detect buffer overflow (prior to the SSA board), if the buffer becomes full (0xfffe samples) the driver assumes data has been lost. The driver will then clear out the hardware buffer and fail the read.

Usage

`ioctl()` Argument	**Description**
`Command`	`IOCTL_DEVICE_DETECT_OVERFLOW`
`Arg`	`unsigned long *pSetting`

4.5.34. IOCTL_DEVICE_SET_UNIPOLAR_INPUTS_SSA

`IOCTL_DEVICE_SET_UNIPOLAR_INPUTS_SSA` is used to select either bipolar or unipolar inputs.

Possible values are:

```
BIPOLAR_INPUTS
UNIPOLAR_INPUTS
```

Usage

`ioctl()` Argument	Description
`command`	`IOCTL_DEVICE_SET_UNIPOLAR_INPUTS_SSA`
`arg`	`unsigned long *pSetting`

IOCTL_DEVICE_SET_DIFF_PROCESSING_SSA

`IOCTL_DEVICE_SET_DIFF_PROCESSING_SSA` is used to set the mode used for differential processing.

Possible values are:

```
SINGLE_ENDED_PROC
PSEUDO_DIFFERETIAL_PROC
FULL_DIFFERENTIAL_PROC
```

Usage

`ioctl()` Argument	Description
`command`	`IOCTL_DEVICE_SET_DIFF_PROCESSING_SSA`
`arg`	`unsigned long *pSetting`

4.5.35. IOCTL_DEVICE_READ_BUFFER_UNDERFLOW_SSA

`IOCTL_DEVICE_READ_BUFFER_UNDERFLOW_SSA` is used to read the status of the buffer underflow bit in the BCR register. Returns non-zero to indicate a buffer underflow. The state of the bit is not reset by this IOCTL. Use `IOCTL_DEVICE_RESET_BUFFER_UNDERFLOW_SSA` to reset the bit.

Usage

`ioctl()` Argument	Description
`command`	`IOCTL_DEVICE_READ_BUFFER_UNDERFLOW_SSA`
`arg`	`unsigned long *pState`

4.5.36. IOCTL_DEVICE_READ_BUFFER_OVERFLOW_SSA

`IOCTL_DEVICE_READ_BUFFER_OVERFLOW_SSA` is used to read the status of the buffer overflow bit in the BCR register. Returns non-zero to indicate a buffer overflow. The state of the bit is not reset by this IOCTL. Use `IOCTL_DEVICE_RESET_BUFFER_OVERFLOW_SSA` to reset the bit.

Usage

`ioctl()` **Argument**	**Description**
`command`	`IOCTL_DEVICE_READ_BUFFER_OVERFLOW_SSA`
`arg`	`unsigned long *pState`

4.5.37. IOCTL_DEVICE_SET_SCAN_MARKING_SSA

`IOCTL_DEVICE_SET_SCAN_MARKING_SSA` is used to enable or disable scan marking and data packing. See the hardware manual for details about what scan marking and data packing are and how they are used.

Possible values are:
`TRUE` – Enable scan marking and data packing.
`FALSE` – Disable scan marking and data packing.

Usage

`ioctl()` **Argument**	**Description**
`command`	`IOCTL_DEVICE_SET_SCAN_MARKING_SSA`
`arg`	`unsigned long *pSetting`

4.5.38. IOCTL_DEVICE_SET_DEMAND_MODE_DISABLE

`IOCTL_DEVICE_SET_DEMAND_MODE_DISABLE_SSA` is used to disable Demand Mode DMA transfers.

Possible values are:
`TRUE` – Disable Demand Mode DMA.
`FALSE` – Allow Demand Mode DMA to be selected.

Usage

`ioctl()` **Argument**	**Description**
`command`	`IOCTL_DEVICE_SET_DEMAND_MODE_DISABLE_SSA`
`arg`	`unsigned long *pSetting`

4.5.39. IOCTL_DEVICE_SET_THRESHOLD_4X_SSA

`IOCTL_DEVICE_SET_THRESHOLD_4X_SSA` is used to enable or disable the 4x multiplier for the buffer threshold register. This feature is useful when the desired threshold value is larger than the range allowed by the buffer threshold register.

Possible values are:
`TRUE` – Enable the 4x threshold level.
`FALSE` – Disable the 4x threshold level.

Usage

`ioctl()` Argument	**Description**
`command`	`IOCTL_DEVICE_SET_THRESHOLD_4X_SSA`
`arg`	`unsigned long *pSetting`

4.5.40. IOCTL_DEVICE_RESET_BUFFER_UNDERFLOW_SSA

`IOCTL_DEVICE_RESET_BUFFER_UNDERFLOW_SSA` is used to reset the underflow bit in the BCR register. There are no arguments passed to this IOCTL.

Usage

`ioctl()` Argument	**Description**
`command`	`IOCTL_DEVICE_RESET_BUFFER_UNDERFLOW_SSA`
`arg`	`None`

4.5.41. IOCTL_DEVICE_RESET_BUFFER_OVERFLOW_SSA

`IOCTL_DEVICE_RESET_BUFFER_OVERFLOW_SSA` is used to reset the overflow bit in the BCR register. There are no arguments passed to this IOCTL.

Usage

`ioctl()` Argument	**Description**
`command`	`IOCTL_DEVICE_RESET_BUFFER_OVERFLOW_SSA`
`arg`	`None`

4.5.42. IOCTL_DEVICE_SET_AUX_0_MODE_SSA

`IOCTL_DEVICE_SET_AUX_0_MODE_SSA` is used to set the mode of the channel 0 auxiliary I/O port.

Possible values are:

```
AUX_INACTIVE
AUX_ACTIVE_IN
AUX_ACTIVE_OUT
```

Usage

`ioctl()` Argument	**Description**
`command`	`IOCTL_DEVICE_SET_AUX_0_MODE_SSA`
`arg`	`unsigned long *pSetting`

4.5.43. IOCTL_DEVICE_SET_AUX_1_MODE_SSA

`IOCTL_DEVICE_SET_AUX_1_MODE_SSA` is used to set the mode of the channel 1 auxiliary I/O port.

Possible values are:

```
AUX_INACTIVE
AUX_ACTIVE_IN
AUX_ACTIVE_OUT
```

Usage

`ioctl()` Argument	**Description**
`command`	`IOCTL_DEVICE_SET_AUX_1_MODE_SSA`
`arg`	`unsigned long *pSetting`

4.5.44. IOCTL_DEVICE_SET_AUX_2_MODE_SSA

`IOCTL_DEVICE_SET_AUX_2_MODE_SSA` is used to set the mode of the channel 2 auxiliary I/O port.

Possible values are:

```
AUX_INACTIVE
AUX_ACTIVE_IN
AUX_ACTIVE_OUT
```

Usage

`ioctl()` Argument	**Description**
`command`	`IOCTL_DEVICE_SET_AUX_2_MODE_SSA`
`arg`	`unsigned long *pSetting`

4.5.45. IOCTL_DEVICE_SET_AUX_3_MODE_SSA

`IOCTL_DEVICE_SET_AUX_3_MODE_SSA` is used to set the mode of the channel 3 auxiliary I/O port.

Possible values are:

```
AUX_INACTIVE
AUX_ACTIVE_IN
AUX_ACTIVE_OUT
```

Usage

`ioctl()` Argument	**Description**
`command`	`IOCTL_DEVICE_SET_AUX_3_MODE_SSA`
`arg`	`unsigned long *pSetting`

4.5.46. IOCTL_DEVICE_SET_INPUT_INVERT_MODE_SSA

`IOCTL_DEVICE_SET_INPUT_INVERT_MODE_SSA` is used to determine if the input transition is triggered on the rising edge or falling edge of the event.

Possible values are:

`TRUE` – Inputs are detected on the high-to-low transition.
`FALSE` – Inputs are detected on low-to-high transition.

Usage

`ioctl()` Argument	**Description**
`command`	`IOCTL_DEVICE_SET_INPUT_INVERT_MODE_SSA`
`arg`	`unsigned long *pSetting`

4.5.47. IOCTL_DEVICE_SET_OUTPUT_INVERT_SSA

`IOCTL_DEVICE_SET_OUTPUT_INVERT_MODE_SSA` is used to select inverted or non-inverted output pulses.

Possible values are:

`TRUE` – Active outputs produce high output pulse.
`FALSE` – Active outputs produce low output pulse.

Usage

`ioctl()` Argument	**Description**
`command`	`IOCTL_DEVICE_SET_OUTPUT_INVERT_MODE_SSA`
`arg`	`unsigned long *pSetting`

4.5.48. IOCTL_DEVICE_SET_NOISE_SUPPRESSION_SSA

`IOCTL_DEVICE_SET_NOISE_SUPPRESSION_SSA` is used to enable or disable input and output noise suppression. See the hardware manual for a description of noise suppression.

Possible values are:

`TRUE` – Enable noise suppression.
`FALSE` – Disable noise suppression.

Usage

`ioctl()` Argument	**Description**
`command`	`IOCTL_DEVICE_SET_NOISE_SUPPRESSION_SSA`
`arg`	`unsigned long *pSetting`

4.5.49. IOCTL_DEVICE_SET_HW_BUFFER_SIZE

`IOCTL_DEVICE_SET_HW_BUFFER_SIZE` is used to tell the driver the size of the hardware FIFO buffer. The buffer size is used by the driver to detect possible data overflow. It does not impact the effective size of the hardware buffer. The default is 64K.

Possible values are:
0-MAX_BUFFER_SIZE (512 K)

Usage

`ioctl()` Argument	**Description**
`command`	`IOCTL_DEVICE_SET_HW_BUFFER_SIZE`
`arg`	`unsigned long *pSetting`

Document History

Revision	**Description**
October 2, 2003	Initial release.
November 25, 2003	Corrected the product description for the 12AI64.
December 12, 2003	Added IOCTL_DEVICE_SET_TIMEOUT Added two missing SS only registers. Corrected the description of the test program output.
January 21, 2004	Added `IOCTL_DEVICE_FILL_BUFFER`, `IOCTL_DEVICE_DETECT_OVERFLOW` and updated the description of the test program.
September 13, 2004	Added referenced to support for kernel 2.6.
May 2, 2005	Added new IOCTLs to support the SSA board. Fixed some typos.
August 30, 2005	Added IOCTL to set hardware buffer size. Added fixes for DMA and interrupt handling on the 2.6 kernel.