ADADIO

16-bit, 12 Channel Analog Input/Output Board

PMC-ADADIO

Linux Device Driver
User Manual

Manual Revision: November 3, 2004

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

Table of Contents

1. Introduction

1.1. Purpose

The purpose of this document is to describe the software interface to the ADADIO Linux device driver. The driver software provides the interface between Application Software and the ADADIO board. The driver is supplied with a sample test application.

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
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 ADADIO driver software executes under control of the Linux operating system and runs in Kernel Mode as a Kernel Mode device driver. The ADADIO device driver is implemented as a standard dynamically loadable Linux device driver written in the C programming language. The driver allows user applications to: open, close, read, write and perform I/O control operations.

1.5. Hardware Overview

See the hardware manual for the board version for details on the hardware. Current board manual PDF files may be found at:

http://www.generalstandards.com/

Look under the "device user manuals" heading and select your board model.

1.6. Reference Material

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

- The applicable *ADADIO 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 2.4 and 2.6 running on a PC system with Intel x86 processor(s). Testing was performed under Red Hat Linux with kernel version 2.4.18-14 and version 2.6.8-1.521smp on a PC system with dual Intel x86 processors. Support for version 2.2 of the kernel has been left in the driver, but has not been tested.

NOTES:

- The driver will probably have to be rebuilt before being used due to kernel version differences between the GSC build host and the customer's target host.
- The driver has not been tested with a non-versioned kernel.
- The driver has only been tested on an SMP host. SMP testing is much more rigorous than single CPU systems, and helps to ensure reliability on single CPU systems.

2.2. The /proc File System

While the driver is installed, the text file `/proc/adadio` 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. Note that with a debug build, there may be more information in the file.

```
version: 2.00
built: Oct 28 2004, 09:08:07

boards: 1
```

Entry	Description
`Version`	The driver version number in the form `x.xx`.
`Built`	The drivers build date and time as a string. It is given in the C form of `printf("%s, %s", __DATE__, __TIME__)`.
`Boards`	The total number of boards the driver detected.

2.3. File List

See the README.TXT file in the release tar for the latest file list.

This section discusses unpacking, building, installing and running the driver.

2.3.1. Installation

Install the driver and its related files following the below listed steps.

1. Create and change to the directory where you would like to install the driver source, such as `/usr/src/linux/drivers`.
2. Copy the `gsc_adadio.tar.gz` file into the current directory. The actual name of the file may be different depending on the release version.

3. Issue the following command to decompress and extract the files from the provided archive. This creates the directory `gsc_adadio_release` in the current directory, and then copies all of the archive's files into this new directory.

```
tar –xzvf gsc_adadio.tar.gz
```

2.3.2. Build

To build the driver:

1. Change to the directory where the driver and its sources were installed in the previous step. Remove all existing build targets by issuing the command:

```
make clean
```

2. Edit Makefile to ensure that the KERNEL_DIR environment variable points to the correct root of the source tree for your version on Linux. The driver build uses different header versions than an application build, which is why this step is necessary. The default should be correct for 2.4 and newer kernels.

3. Build the driver by issuing the command:

```
make all
```

NOTE: Due to the differences between the many Linux distributions some build errors may occur. The most likely cause is not having the kernel sources installed properly. See the documentation for your release of Linux for instructions on how to install the kernel sources.

To build the test applications:

1. Type the command:

```
make –f app.mak
```

2.3.3. Startup

The startup script used in this procedure is designed to ensure 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 corresponds to the number of boards identified by the driver.

2.3.3.1. 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 was installed. In this example, this would be `/usr/src/linux/drivers/gsc_adadio_release.`

3. Type:

```
./gsc_start
```

The script assumes that the driver be installed in the same directory as the script, and that the driver filename has not been changed from that specified in Makefile. The above step must be repeated each time the host is rebooted. It is possible to have the script run at system startup. See below for instructions on automatically starting the driver.

NOTE: The kernel assigns the ADADIO device node major number dynamically. 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 module has been loaded by issuing the below command and examining the output. The module name `adadio` 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/adadio*
```

2.3.3.2. Automatic Driver Startup Procedures

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

1. Locate and edit the system startup script `rc.local`, which should be in the `/etc/rc.d` directory. Modify the file by adding the below line so that it is executed with every reboot.

```
/usr/src/linux/drivers/gsc_adadio_release/gsc_start
```

NOTE: The script assumes the driver is in the same directory as the script. Change the path as required to point to the actual location of the driver.

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.

2.3.4. Verification

To verify that the hardware and driver are installed properly and working, the steps are:

1. Install the sample applications, if they were not installed as part of the driver install.

2. Change to the directory where the sample application `testapp` was installed.

3. Start the sample application by issuing the below command. The argument identifies which board to access. The argument is the zero based index of the board to access.

```
./testapp <board>
```

So for a single-board installation, type:

```
./testapp 0
```

The test application is described in greater detail in a later section.

2.3.5. Version

The driver version number can be obtained in a variety of ways. It is appended to the system log when the driver is loaded or unloaded (type `dmesg` to view the contents of the system log file). It is recorded in the text file `/proc/adadio`. It is also in the driver source header file `internals.h`, which is where the version number is maintained.

2.3.6. 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 adadio
   ```

3. Verify that the driver module has been unloaded by issuing the below command. The module name `adadio` should not be in the list.

   ```
   lsmod
   ```

2.3.7. Removal

Follow the below steps to remove the driver.

1. Shutdown the driver as described in the previous paragraphs.

2. Change to the directory where the driver archive was installed. This should be `/usr/src/linux/drivers`.

3. Issue the below command to remove the driver archive and all of the installed driver files.

   ```
   rm –rf adadio.tar.gz gsc_adadio_release
   ```

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

   ```
   rm –f /dev/adadio*
   ```

5. If the automated startup procedure was adopted, then edit the system startup script `rc.local` and remove the line that invokes the `gsc_start` script. The file `rc.local` should be located in the `/etc/rc.d` directory.

2.4. Sample Application

The archive file `gsc_adadio.tar.gz` contains a sample application. The test application is a Linux user mode application whose purpose is to demonstrate the functionality of the driver with an installed board. They are delivered undocumented and unsupported. They can however be used as a starting point for developing applications on top of the Linux driver and to help ease the learning curve. The principle application is described in the following paragraphs.

2.4.1. testapp

This sample application provides a command line driven Linux application that tests the functionality of the driver and a user specified ADADIO board. It can be used as the starting point for application development on top of the ADADIO Linux device driver. The application performs an automated test of the driver features. The application includes the below listed files.

File	Description
`testapp.c`	The test application source file.
`testapp`	The pre-built sample application.
`app.mak`	The build script for the sample application.

2.4.2. Installation

The test application is normally installed as part of the driver install, in the same directory as the driver.

2.4.3. Build

The test applications require different header files than the driver, consequently they require a separate make script. Follow the below steps to build/rebuild the sample application.

1. Change to the directory where the sample application was installed.

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

   ```
   make –f app.mak clean
   ```

3. Build the sample applications by issuing the below command.

   ```
   Make –f app.mak
   ```

NOTE: The build procedure assumes the driver header files are located in the current directory.

2.4.4. Execute

Follow the below steps to execute the sample application.

1. Change to the directory where the sample application was installed.

2. Start the sample application by issuing the command given below. The argument specifies the index of the board to access. Use 0 (zero) if only one board is installed.

   ```
   ./testapp 0
   ```

2.4.5. Removal

```
The sample application is removed when the driver is removed.
```

3. Driver Interface

The ADADIO 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 uniform driver interface to the ADADIO 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 ADADIO specific portion of the driver interface is defined in the header file `adadio_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.

3.1. Macros

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

3.1.1. IOCTL

The IOCTL macros are the primary means to change the settings and configuration of the hardware. The IOCTLs are documented following the function call descriptions.

3.1.2. Registers

The driver allows access to the local ADADIO registers, but not the PCI configuration registers. Normally the PCI configuration registers do not require modification.

3.1.2.1. GSC Registers

The following table gives the complete set of GSC specific ADADIO registers. For detailed definitions of these registers refer to the relevant ADADIO User Manual. The macro defines of the registers are located in `adadio_ioctl.h`. Note that the hardware manual defines the register address in 8-bit address space. The driver maps the registers in 32-bit space. For example, the `ANALOG_INPUT_REG` register has local address 0x18 as defined in the hardware manual. The driver accesses this register at local address 6 (0x18/4).

BOARD_CTRL_REG
DIGITAL_IO_PORT_REG
ANALOG_OUTPUT_CHAN0_REG
ANALOG_OUTPUT_CHAN1_REG
ANALOG_OUTPUT_CHAN2_REG
ANALOG_OUTPUT_CHAN3_REG
ANALOG_INPUT_REG
SAMPLE_RATE_REG

3.2. Data Types

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

3.2.1. device_register_params

This structure is used to transfer register data. The IOCTL_GSC_READ_REGISTER and IOCTL_GSC_WRITE_REGISTER ioctls use this structure to read and write a user selected register. 'eRegister' stores the index of the register, range 0-LAST_LOCAL_REGISTER, and ‘ulValue’ stores the register value being written or read. The absolute range for ‘ulValue’ is 0x0-0xFFFFFFFF, and the actual range depends on the register accessed.

Definition

```
typedef struct device_register_params {
    __u32 eRegister;
    __u32 ulValue;
} DEVICE_REGISTER_PARAMS, *PDEVICE_REGISTER_PARAMS;
```

Fields	Description
`eRegister`	Register to read or write. See `adadio_ioctl.h` for register definitions.
`ulValue`	Value read from, or written to above register.

3.3. Functions

This driver interface includes the following functions.

3.3.1. open()

This function is the entry point to open a handle to a ADADIO board.

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 <unistd.h>
#include “adadio_ioctl.h”

int adadio_open(unsigned int board)
{
```

```
        int     fd;
        char    name[80];

        sprintf(name, "/dev/adadio%u", board);
        fd  = open(name, O_RDWR);

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

        return(fd);
    }
```

3.3.2. read()

The `read()` function is used to retrieve data from the hardware input buffer. The application passes down the handle of the driver instance, a pointer to the user 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. Use `IOCTL_GSC_SET_FILL_INPUT_BUFFER` to force the driver to fill the user buffer before returning.

Linux does not support a simple way to transfer data directly from hardware to the user buffer. Therefore the driver uses an intermediate buffer to transfer data from the hardware, and to the user buffer.

How the intermediate 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 the intermediate 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 then 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**: Not all ADADIO boards support demand-mode DMA. Please refer to your hardware manual.

DMA always uses an intermediate system buffer then copies the resulting data into the user buffer. It is not currently possible with (as of versions 2.2 through 2.6) Linux to DMA directly into a 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 internals.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.

Prototype

```
int read(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 read. 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. 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 when the receive FIFO became empty during the request.

Example:

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

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

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

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

    return(status);
}
```

3.3.3. write()

```
The write() function is not used by the ADADIO.  Writes to the hardware
will return an error.
```

3.3.4. close()

Close the handle to the device.

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 <unistd.h>
#include “adadio_ioctl.h”

int adadio_close(int fd)
{
    int status;

    status  = close(fd);

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

    return(status);
}
```

3.4. IOCTL Services

This function is the entry point to performing setup and control operations on an ADADIO 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_GSC_READ_REGISTER` to read the contents of the board control register (BCR):

```
#include "adadio_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_GSC_READ_REGISTER, &regdata);
    if (res < 0) {
        printf("%s: ioctl IOCTL_GSC_READ_REGISTER failed\n", argv[0]);
        }
    return (res);
```

3.4.1. IOCTL_GSC_NO_COMMAND

NO-OP call. IOCTL_GSC_NO_COMMAND is useful for verifying that the board has been opened properly.

Usage

`ioctl()` Argument	Description
Request	IOCTL_GSC_NO_COMMAND

3.4.2. IOCTL_GSC_READ_REGISTER

This service reads the value of an ADADIO register. This includes all GSC specific registers. Refer to `adadio_ioctl.h` for a complete list of the accessible registers.

Usage

`ioctl()` Argument	Description
Request	IOCTL_GSC_READ_REGISTER
Arg	device_register_params*

3.4.3. IOCTL_GSC_WRITE_REGISTER

This service writes a value to an ADADIO register. This includes only the GSC specific registers. Refer to `adadio_ioctl.h` for a complete list of the accessible registers.

Usage

`ioctl()` Argument	Description
Request	IOCTL_GSC_WRITE_REGISTER

Arg	device_register_params*

3.4.4. IOCTL_GSC_GET_DEVICE_ERROR

This call is used to retrieve the detailed error code for the most recent error. Possible return values are:

```
ADA_SUCCESS
ADA_INVALID_PARAMETER
ADA_INVALID_BUFFER_SIZE
ADA_PIO_TIMEOUT
ADA_DMA_TIMEOUT
ADA_IOCTL_TIMEOUT
ADA_OPERATION_CANCELLED
ADA_RESOURCE_ALLOCATION_ERROR
ADA_INVALID_REQUEST
```

Usage

ioctl() Argument	**Description**
Request	IOCTL_GSC_GET_DEVICE_ERROR
Arg	unsigned long*

3.4.5. IOCTL_GSC_SET_TIMEOUT

Set the timeout for reading, writing, autocalibration and initialization, in seconds. Range 0-0xFFFFFFFF. Default is 5 seconds.

Usage

ioctl() Argument	**Description**
Request	IOCTL_GSC_SET_TIMEOUT
Arg	unsigned long *

3.4.6. IOCTL_GSC_SET_DMA_MODE

Used to select if DMA is enabled, disabled or demand-mode. Choices are:

```
DMA_DISABLE
DMA_ENABLE
DMA_DEMAND_MODE
```

For most systems DMA is the preferred choice. Default is `DMA_DISABLE`. Note that only newer ADADIO hardware will support demand mode. Check your hardware manual for details.

Usage

ioctl() Argument	**Description**
Request	IOCTL_GSC_SET_DMA_MODE
Arg	unsigned long *

3.4.7. IOCTL_GSC_CONFIG_INPUTS

Set the input mode. Possible values are:

```
SINGLE_ENDED_CONTINUOUS
SINGLE_ENDED_BURST
DIFFERENTIAL_CONTINUOUS
DIFFERENTIAL_BURST
LOOPBACK_SELFTEST
VREF_TEST
ZERO_TEST
```

Usage

`ioctl()` Argument	Description
Request	IOCTL_GSC_CONFIG_INPUTS
Arg	unsigned long *

3.4.8. IOCTL_GSC_SELECT_LOOPBACK_CHANNEL

Set the input voltage range. Range 0 to 3, selecting channel 0 through 3.

Usage

`ioctl()` Argument	Description
Request	IOCTL_GSC_SELECT_LOOPBACK_CHANNEL
Arg	unsigned long*

3.4.9. IOCTL_GSC_CALIBRATE

Initiate an auto-calibrate cycle.

Usage

`ioctl()` Argument	Description
Request	IOCTL_GSC_CALIBRATE
Arg	None

3.4.10. IOCTL_GSC_SET_DATA_FORMAT

Sets the analog input data format.   Possible values are:

```
FORMAT_TWOS_COMPLEMENT
FORMAT_OFFSET_BINARY
```

Usage

`ioctl()` Argument	Description
Request	IOCTL_GSC_SET_DATA_FORMAT
Arg	unsigned long *

3.4.11. IOCTL_GSC_SET_INPUT_BUFFER_SIZE

Set the virtual size of the input buffer, in samples.   Possible values are:

```
BUF_SIZE_1
BUF_SIZE_2
BUF_SIZE_4
BUF_SIZE_8
BUF_SIZE_16
BUF_SIZE_32
BUF_SIZE_64
BUF_SIZE_128
BUF_SIZE_256
BUF_SIZE_512
BUF_SIZE_1024
BUF_SIZE_2048
BUF_SIZE_4096
BUF_SIZE_8192
BUF_SIZE_16384
BUF_SIZE_32768
```

Usage

ioctl() Argument	**Description**
Request	IOCTL_GSC_SET_INPUT_BUFFER_SIZE
Arg	unsigned long *

3.4.12. IOCTL_GSC_ENABLE_OUTPUTS

Enable or disable analog outputs. Use the argument TRUE to enable the outputs, and FALSE to disable the outputs. Possible values are:

```
TRUE
FALSE
```

Usage

Ioctl() Argument	**Description**
Request	IOCTL_GSC_ENABLE_OUTPUTS
Arg	unsigned long *

3.4.13. IOCTL_GSC_ENABLE_STROBE

Enable or disable output strobe. Use the argument TRUE to enable the strobe, and FALSE to disable the strobe. Possible values are:

```
TRUE
FALSE
```

Usage

Ioctl() Argument	**Description**
Request	IOCTL_GSC_ENABLE_STROBE
Arg	unsigned long *

3.4.14. IOCTL_GSC_STROBE_OUTPUTS

Initiates an output strobe cycle.

Usage

ioctl() Argument	Description
Request	IOCTL_GSC_STROBE_OUTPUTS

3.4.15. IOCTL_GSC_TRIGGER_INPUTS

Triggers input sampling.

Usage

ioctl() Argument	Description
Request	IOCTL_GSC_TRIGGER_INPUTS

3.4.16. IOCTL_GSC_INITIALIZE

Initialize the board to a known state. Sets all defaults. The driver waits for an interrupt from the hardware indicating that the initialization cycle is complete.

Usage

ioctl() Argument	Description
Request	IOCTL_GSC_INITIALIZE

3.4.17. IOCTL_GSC_SET_DIO_DIR

Set the direction of the digital I/O bits 0-7. Possible values are:

```
DIO_DIR_INPUT
DIO_DIR_OUTPUT
```

Usage

ioctl() Argument	Description
Request	IOCTL_GSC_SET_DIO_DIR
Arg	Unsigned long *

3.4.18. IOCTL_GSC_SET_DIO

Sets the DIO port to the passed value. Range is 0 to 255 (DIO_MAX_VALUE). Assumes the DIO port is set to output.

Usage

Ioctl() Argument	Description
request	IOCTL_GSC_SET_DIO
Arg	unsigned long *

3.4.19. IOCTL_GSC_GET_DIO

Reads the DIO port. Range is 0 to 255 (DIO_MAX_VALUE). Assumes the DIO port is set to input.

Usage

`ioctl()` Argument	Description
request	IOCTL_GSC_GET_DIO
arg	unsigned long *

3.4.20. IOCTL_GSC_SET_DIO_CTRL

Set the DIO dedicated output bit to high or low. Use TRUE to set the bit. Use FALSE to reset the bit. Possible values are:

```
TRUE
FALSE
```

Usage

`ioctl()` Argument	Description
request	IOCTL_ADS_SET_DIO_CTRL
arg	unsigned long *

3.4.21. IOCTL_GSC_GET_DIO_CTRL

Reads the DIO dedicated input bit. Returns 1 or 0.

```
0
1
```

Usage

`ioctl()` Argument	Description
request	IOCTL_GSC_GET_DIO_CTRL
arg	unsigned long *

3.4.22. IOCTL_GSC_SET_NRATE

Sets the NRATE register to set the clock speed. Range is 100-65535 (0x100-0xFFFF).

Usage

`ioctl()` Argument	Description
request	IOCTL_GSC_SET_NRATE
arg	unsigned long *

3.4.23. IOCTL_GSC_GET_DEVICE_TYPE

Returns the type of device. Useful when one driver supports multiple variants of the same board. Currently the ADADIO only has one variant. Returns 0.

Returns:

0

Usage

ioctl() Argument	Description
request	IOCTL_GSC_GET_DEVICE_TYPE
arg	Unsigned long *

3.4.24. IOCTL_GSC_WRITE_ANALOG_0

Writes the passed value to analog output channel 0. Range is 0 to 0xFFFF (MAX_ANALOG_OUT).

Usage

ioctl() Argument	Description
request	IOCTL_GSC_WRITE_ANALOG_O
arg	unsigned long *

3.4.25. IOCTL_GSC_WRITE_ANALOG_1

Writes the passed value to analog output channel 1. Range is 0 to 0xFFFF (MAX_ANALOG_OUT).

Usage

ioctl() Argument	Description
request	IOCTL_GSC_WRITE_ANALOG_1
arg	unsigned long *

3.4.26. IOCTL_GSC_WRITE_ANALOG_2

Writes the passed value to analog output channel 2. Range is 0 to 0xFFFF (MAX_ANALOG_OUT).

Usage

ioctl() Argument	Description
request	IOCTL_GSC_WRITE_ANALOG_2
arg	unsigned long *

3.4.27. IOCTL_GSC_WRITE_ANALOG_3

Writes the passed value to analog output channel 3. Range is 0 to 0xFFFF (MAX_ANALOG_OUT).

Usage

ioctl() Argument	Description
request	IOCTL_GSC_WRITE_ANALOG_3
arg	unsigned long *

3.4.28. IOCTL_GSC_CLEAR_BUFFER

Used to remove all data from the analog input buffer. No argument.

Usage

`ioctl()` Argument	Description
`request`	`IOCTL_GSC_CLEAR_BUFFER`
`arg`	`none`

3.4.29. IOCTL_GSC_FILL_INPUT_BUFFER

This ioctl is used to instruct the driver to fill the user buffer before returning. If set TRUE, the driver will make one or more read transfers from the hardware to satisfy the user request. If the state is set to FALSE, the driver will return one or more samples per the Linux convention. Default is FALSE.

Usage

`ioctl()` Argument	Description
`request`	`IOCTL_GSC_FILL_INPUT_BUFFER`
`arg`	`unsigned long *`

4. Operation

This section explains some operational procedures using the driver. This is in no way intended to be a comprehensive guide on using the ADADIO. This is simply to address a few issues relating to using the ADADIO.

4.1. Read Operations

Before performing `read()` requests the device I/O parameters should be configured via the appropriate IOCTL services.

4.2. Data Reception

Data reception is essentially a three-step process; configure the ADADIO, initiate data conversion and read the converted data. A simplified version of this process is illustrated in the steps outlined below.

1. Perform a board reset to put the ADADIO in a known state.

2. Perform the steps required for any desired input voltage range, number of channels, scan rate settings, etc.

3. Initiate a date conversion cycle.

4. Use the `read()` service to retrieve the data from the board.

4.3. Data Transfer Options

4.3.1. PIO

This mode uses repetitive register accesses in performing data transfers and is most applicable for low throughput requirements.

4.3.2. Standard DMA

This mode is intended for data transfers that do not exceed the size of the ADADIO data buffer. In this mode, all data transfer between the PCI interface and the data buffers is done in burst mode. The data must be in the hardware buffer before the DMA transfer will start.

Document History

Revision	Description
November 2, 2004	Initial draft.