24DSI32PLL

24-bit, 32 Channel Delta-Sigma A/D Board with Phase Locked Loop Frequency Generation

GSC-24DSI32PLL

Linux Device Driver
User Manual

Manual Revision: February 19, 2007

General Standards Corporation
8302A Whitesburg Drive
Huntsville, AL 35802

Phone: (256) 880-8787
Fax: (256) 880-8788

URL: http://www.generalstandards.com
E-mail: sales@generalstandards.com
E-mail: support@generalstandards.com

Preface

Copyright ©2004-2007, **General Standards Corporation**

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

General Standards Corporation
8302A Whitesburg Dr.
Huntsville, Alabama 35802
Phone: (256) 880-8787
FAX: (256) 880-8788
URL: http://www.generalstandards.com
E-mail: sales@generalstandards.com

General Standards Corporation makes no warranty of any kind with regard to this 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.

ALL RIGHTS RESERVED.

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 24DSI32PLL Linux device driver. The driver software provides the interface between "Application Software" and the 24DSI32PLL board. The designation "24DSI32PLL" is used throughout the document to refer to any member of the board family, including the PCI-24DSI32PLL and cPCI6U-24DSI32R.

1.2. Definitions and Acronyms

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.
DMA	Direct Memory Access
GSC	General Standards Corporation
PCI	Peripheral Component Interconnect
PMC	PCI Mezzanine Card

1.3. Software Overview

The 24DSI32PLL driver software executes under control of the Linux operating system and runs in Kernel Mode as a Kernel Mode device driver. The 24DSI32PLL 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, and perform I/O control operations. Data write to the hardware is not supported.

1.4. 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.

1.5. Reference Material

The following reference material may be of particular benefit in using the 24DSI32PLL and this driver. The specifications provide the information necessary for an in depth understanding of the specialized features implemented on this board.

- The applicable *24DSI32PLL* or *24DSI12User 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 versions 2.4.18-14 and 2.6.xsmp 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 may 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/gsc24dsi32pll` 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: 1.0.4
built: June 13 2006, 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.

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

2.4.1. 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_24DSI32.tar.gz` file into the current directory. The actual name of the file will 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_24DSI32PLL_release` in the current directory, and then copies all of the archive's files into this new directory.

   ```
   tar –xzvf gsc_24DSI32PLLDriver.tar.gz
   ```

2.4.2. 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 below 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 below command.

   ```
   make
   ```

   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
   ```

2.4.3. 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.

2.4.3.1. 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_24DSI32PLL_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 24DSI32PLL 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 `gsc24dsi32pll` 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/gsc24dsi32pll*
```

2.4.3.2. 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_24DSI32PLL_release/gsc_start
```

NOTE: The script assumes the driver is in the same directory as the script.

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.

2.4.4. 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.

2.4.5. 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. It is recorded in the text file `/proc/gsc24dsi32pll`. It is also in the driver source header file `gsc24dsi32pll.h`,.

2.4.6. 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 gsc24dsi32pll
   ```

3. Verify that the driver module has been unloaded by issuing the below command. The module name `gsc24dsi32pll` should not be in the list.

   ```
   lsmod
   ```

2.4.7. 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 gsc24dsi32pllDriver.tar.gz gsc_24DSI32PLL_release
   ```

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

   ```
   rm –f /dev/gsc24dsi32pll*
   ```

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.

2.5. Sample Application

The archive file 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.

2.5.1. testapp

This sample application provides a command line driven Linux application that tests the functionality of the driver and a user specified 24DSI32PLL board. It can be used as the starting point for application development on top of the 24DSI32PLL 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.

2.5.2. Installation

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

2.5.3. Build

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.

2.5.4. 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
```

2.5.5. Removal

The sample application is removed when the driver is removed.

3. Driver Interface

The 24DSI32PLL 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 24DSI32PLL 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 24DSI32PLL specific portion of the driver interface is defined in the header file `gsc24dsi32pll.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 gsc24dsi32pll`_ioctl.h`. The header also contains various other utility type macros, which are provided without documentation.

3.1.1. IOCTL

The IOCTL macros are documented following the function call descriptions.

3.1.2. Registers

The following tables give the complete set of 24DSI32PLL registers. The tables are divided by register categories. Unless otherwise stated, all registers are accessed as 32-bits. The only exception is the PCICCR register, which is 24-bits wide but accessed as if it were 32-bits wide. In this instance the upper eight-bits are to be ignored. Register values are passed as 32-bit entities and bits outside the register's native size are ignored.

3.1.2.1. GSC Registers

The following table gives the complete set of GSC specific 24DSI32PLL registers. For detailed definitions of these registers refer to the relevant 24DSI32PLL User Manual. The macro defines of the registers are located in gsc24dsi32pll_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 BUFFER_CONTROL register has local address 0x20 as defined in the hardware manual. The driver accesses this register at local address 8 (0x20/4). All this should be transparent to the user program is the #defines in the `gsc24dsi32_ioctl.h` file are used.

BOARD_CTRL_REG
NREF_PLL_CONTROL_REG
NVCO_PLL_CONTROL_REG
RATE_ASSIGN_REG
RATE_DIVISORS_REG
RESERVED_1
PLL_REF_FREQ_REG
RANGE_FILTER_CONTROL
BUFFER_CONTROL_REG
BOARD_CONFIG_REG
BUFFER_SIZE_REG
AUTOCAL_VALUES_REG
INPUT_DATA_BUFFER_REG

3.1.2.2. PCI Configuration Registers

The driver does not allow access to the PLX registers. There is generally no need to modify the PLX registers.

3.2. Data Types

This driver interface includes the following data types, which are defined in `gsc24dsi32pll_ioctl.h`.

3.2.1. board_entry

3.2.2. device_register_params

This structure is used to transfer register data. The IOCTL_GSC_READ_REGISTER and IOCTL_GSC_WRITE_REGISTER use this structure to read and write a user selected register. 'ulRegister' stores the index of the register, range 0-LAST_REG, 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 {
    unsigned int ulRegister;
    unsigned long ulValue;
} DEVICE_REGISTER_PARAMS, *PDEVICE_REGISTER_PARAMS;
```

Fields	Description
`ulRegister`	Register to read or write. See `gsc24dsi32pll_ioctl.h` for register definitions.
`ulValue`	Value read from, or written to above register.

3.2.3. gen_assign_params

The IOCTL_GSC_ASSIGN_GEN_TO_GROUP IOCTL command uses this structure to assign a channel group to a specified generator. 'eGroup' contains the channel group, and 'eGenAssign' specifies which generator.

Definition

```
typedef struct gen_assign_params {
    unsigned int eGroup;
    unsigned int eGenAssign;
} GEN_ASSIGN_PARAMS, *PGEN_ASSIGN_PARAMS;
```

Fields	Description
`eGroup`	The group of channels to use with the selected generator.
`eGenAssign`	The selected generator.

3.3. Functions

This driver interface supports the following functions.

3.3.1. open()

This function is the entry point to open a handle to a 24DSI32PLL board. The pathname is “/dev/gsc24dsi32plln” when ‘n’ is the numerical board index [0, 1, 2…]. For a single board install, the pathname would be “/dev/gsc24dsi32pll0” to access the 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 “gsc24dsi32pll_ioctl.h”

int 24DSI32PLL_open(unsigned int board)
{
    int     fd;
    char    name[80];

    sprintf(name, "/dev/gsc24dsi32pll%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 driver. The application passes down the handle of the driver instance (returned from open()), 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, and 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.

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. Instead, the data must pass through an intermediate DMA-capable buffer. The size of the intermediate buffer is determined by the #define `SG_BUFFER_K_SAMPLES` in the gsc24dsi32pll.h file. The driver allocated as many pages as is required to hold the selected number of samples. The driver uses scatter-gather DMA, which means that the system does not have to allocate the entire buffer as a contiguous block.

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 "gsc24dsi32pll_ioctl.h"

int 24DSI32PLL_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()

This service is not implemented, as the 24DSI32PLL has no destination to which to transfer a block of data. This function will therefore always 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 “gsc24dsi32pll_ioctl.h”

int 24DSI32PLL_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 a 24DSI32PLL 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, unsigned long command);
```

or:

```
int ioct(int fd, unsigned long 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 the following sections.

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 "gsc24dsi32pll_ioctl.h"

int ReadTest(int fd)
{
    device_register_params RegPar;
    unsigned long dwTransferSize;
    int res;

    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 (res);
```

3.4.1. IOCTL_NO_COMMAND

NO-OP call. IOCTL_GSC_NO_COMMAND dumps an image of the board register to the debug log. This is helpful for remote debugging.

Usage

`ioctl()` **Argument**	**Description**
`Request`	`IOCTL_GSC_NO_COMMAND`

3.4.2. IOCTL_READ_REGISTER

This service reads the value of a 24DSI32PLL register. This includes only the GSC specific registers. Refer to `gsc24dsi32pll_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_WRITE_REGISTER

This service writes a value to a 24DSI32PLL register. This includes only the GSC specific registers. Refer to `gsc24dsi32pll_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_SET_INPUT_RANGE

Set the input voltage range. Possible values are:

```
RANGE_2p5V
RANGE_5V
RANGE_10V
```

Usage

`ioctl()` **Argument**	**Description**
`request`	`IOCTL_GSC_SET_INPUT_RANGE`

`arg`	`unsigned long*`

3.4.5. IOCTL_SET_INPUT_MODE

Set the input mode. Possible values are:

```
MODE_DIFFERENTIAL
MODE_ZERO_TEST
MODE_VREF_TEST
```

Usage

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

3.4.6. IOCTL_SET_SW_SYNCH

Initiate an ADC SYNCH operation. Also generates the external sync output if the board is in initiator mode.

Usage

`ioctl()` Argument	**Description**
`request`	`IOCTL_GSC_SET_SW_SYNCH`

3.4.7. IOCTL_AUTO_CAL

Initiate an auto-calibration cycle. Check the hardware manual for what settings should be make before running an autocalibration cycle.

Usage

`ioctl()` Argument	**Description**
`request`	`IOCTL_GSC_AUTO_CAL`

3.4.8. IOCTL_INITIALIZE

Initialize the board to a known state. The initialize operation sets all hardware settings to their 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`

3.4.9. IOCTL_SET_DATA_FORMAT

Set the digital data output format. Options are:

```
FORMAT_TWOS_COMPLEMENT
FORMAT_OFFSET_BINARY
```

Usage

`ioctl()` Argument	Description
`request`	`IOCTL_GSC_SET_DATA_FORMAT`
`arg`	`Unsigned long *`

3.4.10. IOCTL_SET_INITIATOR_MODE

Set this board as the initiator for synchronized acquisition. Options are:

```
TARGET_MODE
INITIATOR_MODE
```

Usage

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

3.4.11. IOCTL_SET_BUFFER_THRESHOLD

Set the data buffer threshold register. The threshold value is used to allow the driver to sleep while waiting for sufficient data for a transfer to the user buffer. Range is 0x0-0x3FFFF (INPUT_BUFFER_SIZE).

Usage

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

3.4.12. IOCTL_CLEAR_BUFFER

Clear any residual data from the data buffer. This command does not halt sampling. For the most consistent results, use `IOCTL_GSC_SET_ACQUIRE_MODE` to halt sampling before clearing the buffer.

Usage

`ioctl()` Argument	Description
`request`	`IOCTL_GSC_CLEAR_BUFFER`

3.4.13. IOCTL_SET_ACQUIRE_MODE

Set the hardware to either start or stop acquiring data. Possible values are:

```
START_ACQUIRE
STOP_ACQUIRE
```

Usage

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

3.4.14. IOCTL_SET_RATE_DIVISOR

Set the value that divides the assigned rate generator frequency (Ndiv). Possible range for Ndiv is NDIV_MIN to NDIV_MAX (0-0xff).

Usage

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

3.4.15. IOCTL_ASSIGN_RATE_GROUP

This ioctl is used to assign a generator to a group consisting of four channels. The group assignments are passed in the structure:

```
typedef struct gen_assign_params {
    __u32 eGroup;                   /* Range: 0-3, see codes below */
    __u32 eGenAssign;               /* Range: 0-6, see codes below */
} GEN_ASSIGN_PARAMS, *PGEN_ASSIGN_PARAMS;
```

Possible values for eGroup are:

```
GRP_0
GRP_1
GRP_2
GRP_3
```

Possible values for eGenAssign are:

```
ASN_RATE_INTERNAL
ASN_EXTERNAL
ASN_DIRECT_EXTERNAL
ASN_DISABLED
ASN_LAST
```

Usage

`ioctl()` Argument	Description
request	IOCTL_ASSIGN_RATE_GROUP
arg	struct gen_assign_params *

3.4.16. IOCTL_SET_NREF

This ioctl is used to set the value of the PLL Nref register. The range is NREF_MIN to NREF_MAX (30-1000).

Usage

`ioctl()` Argument	**Description**
`request`	`IOCTL_GSC_SET_NREF`
`arg`	`Unsigned long *`

3.4.17. IOCTL_SET_NVCO

This ioctl is used to set the value of the PLL voltage controlled oscillator (Nvco) register. The range is NVCO_MIN to NVCO_MAX (30-1000).

Usage

`ioctl()` Argument	**Description**
`request`	`IOCTL_GSC_SET_NVCO`
`arg`	`Unsigned long *`

3.4.18. IOCTL_GET_REF_FREQUENCY

This IOCTL Returns the contents of the PLL frequency reference register. See the hardware manual for the usage of this register.

Usage

`ioctl()` Argument	**Description**
`request`	`IOCTL_GSC_GET_REF_FREQUENCY`
`arg`	`Unsigned long *`

3.4.19. IOCTL_SET_TIMEOUT

Set the wait timeout for reading a data buffer, in seconds. Default is five seconds.

Usage

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

3.4.20. IOCTL_SET_DMA_STATE

Enable or disable DMA for read. Possible values are:

```
DMA_DISABLE
DMA_ENABLE
DMA_DEMAND_MODE
```

For most systems DMA is the preferred choice. Default is `DMA_DISABLE`.

Usage

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

3.4.21. IOCTL_FILL_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_BUFFER`
`arg`	`unsigned long *`

3.4.22. IOCTL_SYNCHRONIZE_SCAN

This IOCTL is used to set the hardware to synchronize scan mode. If set TRUE, the hardware will use synchronize scan mode. If FALSE, the hardware will not use synchronize scan mode. Default is FALSE.

Usage

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

3.4.23. IOCTL_CLEAR_BUFFER_SYNC

This ioctl is used to set the context of the software synch control bit. When TRUE, the software synch control bit becomes "clear buffer."

Usage

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

3.4.24. IOCTL_SET_OVERFLOW_CHECK

This IOCTL is used to enable or disable buffer overflow checking. When TRUE, the driver will check to see if the overflow threshold has been exceeded and fail the read call if it has. If FALSE, the driver ignores the overflow level.

Usage

`ioctl()` **Argument**	**Description**
`Request`	`IOCTL_GSC_SET_OVERFLOW_CHECK`

`Arg`	`unsigned long *`

3.4.25. IOCTL_SELECT_IMAGE_FILTER

This IOCTL is used to select low or high frequency image filtering. Default is high frequency filtering. Possible values are:

```
IMAGE_FILTER_LO_FREQ
IMAGE_FILTER_HI_FREQ
```

Usage

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

3.4.26. IOCTL_SET_DATA_WIDTH

This IOCTL is used to set the bit-width of the output data. Possible values are:

```
DATA_WIDTH_16
DATA_WIDTH_18
DATA_WIDTH_20
DATA_WIDTH_24
```

Usage

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

3.4.27. IOCTL_SET_RANGE_FILTER

This ioctl is used to set the individual channel group low image filter and gain. The entire register is written as a single bit mask.

NOTE: This register is not available on all boards. Check your hardware manual for details.

Usage

`ioctl()` **Argument**	**Description**
`request`	`IOCTL_GSC_SET_RANGE_FILTER`
`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 24DSI32PLL. This is simply to address a few issues relating to using the 24DSI32PLL.

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 24DSI32PLL, 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 24DSI32PLL 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 24DSI32PLL 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.

4.3.3. 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 gsc24dsi32pll.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.

4.4. Data Conversion

The GSC line of data acquisition products uses a somewhat non-standard data encoding scheme. Because a channel tag is included as part of the sample data, the most significant bit cannot be sign-extended to fill the data word.

For example, the hardware may return a data stream like:

```
000027AB
01FFFF06
020023FB
03FFE9C3
04FFF902
05FFF8DA
06000273
07001B9E
08000C9A
09001999
0AFFDE1B
0BFFEEED
0CFFEDBF
0DFFFD58
0EFFF4BD
0FFFED8F
10001679
11FFFAB7
12FFF5D1
13FFE799
14FFED94
150005B2
16FFF4D5
.
.
.
```

In the sequence above, the first two digits are the channel number and the rest is the data value. The channel number must be masked off before conversion.

The next factor is the data resolution. For 16 bit resolution, there is 65535 bits full scale.

The voltage-per-bit depends on the full-scale input range. For the +/-10 volt range, the full scale voltage is 20 volts. So the volts-per-bit would be 20/65535 = 0.30518mV per bit.

Next, the data format must be taken into account. For offset binary, zero would be 0x8000; for two’s complement it would be 0x0000.

For offset-binary:

```
if Reading >= 0x8000
  Voltage = Reading - 0x8000 * .00030518 else
  -Voltage = 0x8000 - Reading * .00030518
```

For 2's Compliment:

```
if Reading >= 0x8000
  -Voltage = -10V + ((Reading - 0x8000) * .00030518) else
  Voltage = Reading * .00030518
```

The algorithm is similar for 18, 20 and 24 bit resolution. For a more complete conversion formula that takes into account all variables, refer to the test application.

Document History

Revision	Description
January 18, 2004	Initial draft.
February 23, 2004	Added new IOCTLs, corrected typographical errors.
July 13, 2004	Added IOCTLs to set buffer overfill level, and enable/disable checking.
August 9, 2004	Added reference to supporting and testing on the 2.6 kernels.
July 7, 2005	Added support for the PMC-24DSI12 and several new IOCTLs.
September 20, 2005	Corrected number of channels reference for the 24DSI12 board from 32 to 12.
February 16, 2007	Updated for PLL devices.