# 24DSI32 – 24DSI12

**24-bit, 32 Channel Delta-Sigma A/D Boards**

## PCI-24DSI32

## PMC-24DSI12

# Linux Device Driver User Manual

**Manual Revision: July 7, 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

# Table of Contents

# 1. Introduction

## 1.1. Purpose

The purpose of this document is to describe the interface to the 24DSI32 Linux device driver. The driver software provides the interface between "Application Software" and the 24DSI32 board. The designation "24DSI32" is used throughout the document to refer to any member of the board family, either the PCI-24DSI32 or the PMC-24DSI12.

## 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 24DSI32 driver software executes under control of the Linux operating system and runs in Kernel Mode as a Kernel Mode device driver. The 24DSI32 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.

## 1.5. 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 24DSI32 and this driver. The specifications provide the information necessary for an in depth understanding of the specialized features implemented on this board.

- The applicable *24DSI32 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 versions 2.4.18-14 and 2.6.7-1.494.2.2smp 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/gsc24dsi32` 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: 1.00
built: June 13 2002, 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. The Driver

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_24dsi32Driver.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_24dsi32_release` in the current directory, and then copies all of the archive's files into this new directory.

   ```
   tar –xzvf gsc_24dsi32Driver.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 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 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_24dsi32_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 24dsi32 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 `gsc24dsi32` 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/gsc24dsi32*
   ```

#### 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_24dsi32_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.

### 2.3.4. 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. It is recorded in the text file `/proc/gsc24dsi32`. It is also in the driver source header file `pci24dsi32.h`,.

### 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 gsc24dsi32
   ```

3. Verify that the driver module has been unloaded by issuing the below command. The module name `gsc24DSI32` 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 gsc24DSI32Driver.tar.gz gsc_24dsi32_release
   ```

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

   ```
   rm –f /dev/gsc24dsi32*
   ```

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`_24dsi32Driver.tar.gz` contains sample applications. The test applications are Linux user mode applications 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 24DSI32 board. It can be used as the starting point for application development on top of the 24DSI32 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 24DSI32 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 24DSI32 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 24DSI32 specific portion of the driver interface is defined in the header file `pci24dsi32.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 `pci24dsi32_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 24DSI32 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 24DSI32 registers. For detailed definitions of these registers refer to the relevant 24DSI32 User Manual. The macro defines of the registers are located in pci24dsi32_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).

| |
|---|
| BOARD_CTRL_REG |
| RATE_CTRL_A_B_REG |
| RATE_CTRL_C_D_REG |
| RATE_ASSIGN_REG |
| RATE_DIVISORS_REG |
| RESERVED_1 |
| RESERVED_2 |
| RESERVED_3 |
| BUFFER_CONTROL_REG |
| BOARD_CONFIG_REG |
| BUFFER_SIZE_REG |
| AUTOCAL_VALUES_REG |
| INPUT_DATA_BUFFER_REG |

#### 3.1.2.2. PCI Configuration Registers

The 16DSDI driver also allows access to the PLX registers. See plx_regs.h for macros defining the registers, and refer to the *PCI9080 Data Book* for detailed descriptions of the registers. Normally, there is no need to access the PLX registers.

## 3.2. Data Types

This driver interface includes the following data types, which are defined in `pci24dsi32_ioctl.h`.

### 3.2.1. board_entry

This structure is used with the `IOCTL_GSC_GET_DEVICE_TYPE` IOCTL.

Definition

```
struct board_entry {
    long subsystem_vendor;
    long subsystem_device;
    char name[40];
    int index;
};
```

| Fields | Description |
|---|---|
| `subsystem_vendor` | The subsystem ID returned from the PCI configuration registers. This value is the same for all General Standards products. |
| `subsystem_device` | The subsystem ID returned from the PCI configuration space. This value is unique for each member of the SDI family. Refer to your hardware manual for the value assigned to your board. |
| `name[40]` | A short string describing the SDI model. |
| `index` | An ENUM `type_index` for board type, defined in pci24dsi32_ioctl.h. |

### 3.2.2. device_register_params

This structure is used to transfer register data. The IOCTL_GSC_READ_REGISTER, IOCTL_GSC_READ_LOCAL_CONFIG_REGISTER, IOCTL_GSC_READ_PCI_CONFIG_REGISTER, IOCTL_GSC_WRITE_REGISTER, IOCTL_GSC_WRITE_PCI_CONFIG_REGISTER and IOCTL_GSC_WRITE_LOCAL_CONFIG_REGISTER IOCTLs use this structure to read and write a user selected register. 'eRegister' stores the index of the register, range 0-94, 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 eRegister;
    unsigned long ulValue;
} DEVICE_REGISTER_PARAMS, *PDEVICE_REGISTER_PARAMS;
```

| Fields | Description |
|---|---|
| `eRegister` | Register to read or write. See `pci24dsi32_ioctl.h` for register definitions. |
| `ulValue` | Value read from, or written to above register. |

### 3.2.3. gen_rate_params

The IOCTL_GSC_SET_GEN_RATE IOCTL uses this structure to set a generator's rate. 'eGenerator' specifies the generator, range, and 'ulNrate' specifies the rate.

Definition

```
 typedef struct gen_rate_params {
    unsigned int eGenerator;
    unsigned long ulNrate;
} GEN_RATE_PARAMS, *PGEN_RATE_PARAMS;
```

| Fields | Description |
|---|---|
| `eGenerator` | Which rate generator to select. |
| `ulNrate` | The rate to use for this generator. |

### 3.2.4. 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. |

### 3.2.5. rate_divisor_params

The IOCTL_GSC_SET_RATE_DIVISOR IOCTL command uses this structure to set the channel group divisor. ulGroup' specifies the channel, and 'ulDivisor' specifies the frequency divisor value. The divisor is calculated using the generator frequency and the sampling rate. See the hardware manual for details.

Definition

```
typedef struct rate_divisor_params {
    unsigned long ulGroup;
    unsigned long ulDivisor;
} RATE_DIVISOR_PARAMS, *PRATE_DIVISOR_PARAMS;
```

| Fields | Description |
|---|---|
| `ulChannel` | The group of channels to use with the selected generator. |
| `ulDivisor` | The divisor for the selected generator. |

### 3.2.6. device_read_pci_config_param

This structure stores information about the PCI Configuration registers. The IOCTL_GSC_READ_PCI_CONFIG IOCTL command uses this structure to read the PCI Configuration registers.

Definition

```
typedef struct device_read_pci_config_param {
    unsigned long ulDeviceVendorID;
    unsigned long ulStatusCommand;
    unsigned long ulClassCodeRevisionID;
    unsigned long ulBISTHdrTypeLatTimerCacheLineSize;
    unsigned long ulRuntimeRegAddr;
    unsigned long ulConfigRegAddr;
    unsigned long ulPCIBaseAddr2;
    unsigned long ulPCIBaseAddr3;
    unsigned long ulUnusedBaseAddr1;
    unsigned long ulUnusedBaseAddr2;
    unsigned long ulCardbusCISPtr;
    unsigned long ulSubsystemVendorID;
    unsigned long ulPCIRomAddr;
    unsigned long ulReserved1;
    unsigned long ulReserved2;
    unsigned long ulMaxLatMinGntIntPinIntLine;
} DEVICE_READ_PCI_CONFIG_PARAM, *PDEVICE_READ_PCI_CONFIG_PARAM;
```

| Fields | Description |
|---|---|
| `ulDeviceVendorID` | Vendor ID |
| `ulStatusCommand` | Status /command |
| `ulClassCodeRevisionID` | Class code, revision |
| `ulBISTHdrTypeLatTimerCacheLineSize` | Latency |
| `ulRuntimeRegAddr` | Runtime registers address (BAR 0) |
| `ulConfigRegAddr` | Unused (BAR 1) |
| `ulPCIBaseAddr2` | Local registers (BAR 2) |
| `ulPCIBaseAddr3` | Unused |
| `ulUnusedBaseAddr1` | Unused |
| `ulUnusedBaseAddr2` | Unused |
| `ulCardbusCISPtr` | Unused |
| `ulSubsystemVendorID` | Device specific ID |
| `ulPCIRomAddr` | Rom address |
| `ulReserved1` | Unused |
| `ulReserved2` | Unused |
| `ulMaxLatMinGntIntPinIntLine` | Latency/grant |

### 3.2.7. config_regs

This structure stores information about all the Local Configuration registers. The IOCTL_GSC_READ_LOCAL_CONFIG IOCTL command uses this structure to read and return all the Local Configuration registers in the PLX.  See the PLX manual for details.

Definition

```
typedef struct config_regs {
    /* ---- Local Configuration Registers ---- */
    unsigned long ulPciLocRange0;
    unsigned long ulPciLocRemap0;
    unsigned long ulModeArb;
    unsigned long ulEndianDescr;
    unsigned long ulPciLERomRange;
    unsigned long ulPciLERomRemap;
    unsigned long ulPciLBRegDescr0;
    unsigned long ulLocPciRange;
    unsigned long ulLocPciMemBase;
    unsigned long ulLocPciIOBase;
    unsigned long ulLocPciRemap;
    unsigned long ulLocPciConfig;
    unsigned long ulOutPostQIntStatus;
    unsigned long ulOutPostQIntMask;
    unsigned char uchReserved1[8];
    /* ---- Shared Run Time Registers ---- */
    unsigned long ulMailbox[8];
    unsigned long ulPciLocDoorBell;
    unsigned long ulLocPciDoorBell;
    unsigned long ulIntCntrlStat;
    unsigned long ulRunTimeCntrl;
    unsigned long ulDeviceVendorID;
    unsigned long ulRevisionID;
    unsigned long ulMailboxReg0;
    unsigned long ulMailboxReg1;
    /* ---- Local DMA Registers ---- */
    unsigned long ulDMAMode0;
    unsigned long ulDMAPCIAddress0;
    unsigned long ulDMALocalAddress0;
    unsigned long ulDMAByteCount0;
    unsigned long ulDMADescriptorPtr0;
    unsigned long ulDMAMode1;
    unsigned long ulDMAPCIAddress1;
    unsigned long ulDMALocalAddress1;
    unsigned long ulDMAByteCount1;
    unsigned long ulDMADescriptorPtr1;
    unsigned long ulDMACmdStatus;
    unsigned long ulDMAArbitration;
    unsigned long ulDMAThreshold;
    unsigned char uchReserved3[12;
    /* ---- Messaging Queue Registers ---- */
    unsigned long ulMsgUnitCfg;
    unsigned long ulQBaseAddr;
    unsigned long ulInFreeHeadPtr;
    unsigned long ulInFreeTailPtr;
    unsigned long ulInPostHeadPtr;
    unsigned long ulInPostTailPtr;
    unsigned long ulOutFreeHeadPtr;
    unsigned long ulOutFreeTailPtr;
    unsigned long ulOutPostHeadPtr;
    unsigned long ulOutPostTailPtr;
    unsigned long ulQStatusCtrl;
    unsigned char uchReserved4[4;
    unsigned long ulPciLocRange1;
    unsigned long ulPciLocRemap1;
    unsigned long ulPciLBRegDescr1;
} CONFIG_REGS, *PCONFIG_REGS;
```

| **Fields** | **Description** |
|---|---|
| `ulPciLocRange0` | range for pci to local 0 |
| `ulPciLocRemap0` | remap for pci to local 0 |
| `UlModeArb` | mode arbitration |
| `UlEndianDescr` | Big/little endian descr. |
| `UlPciLERomRange` | range for pci to local expansion rom |
| `ulPciLBRegDescr0` | Bus region descriptions for pci to local (space 0) |
| `UlLocPciRange` | range for local to pci |
| `ulLocPciMemBase` | base addr for local to pci memory |
| `ulLocPciIOBase` | base addr for local to pci IO/Config |
| `ulLocPciRemap` | remap for local to pci |
| `ulLocPciConfig` | Pci config address reg for local to pci IO/Config |
| `ulOutPostQIntStatus` | outboard post queue interrupt status |
| `ulOutPostQIntMask` | outboard post queue interrupt mask |
| `uchReserved1[8]` | Reserved |
| `ulMailbox[8]` | 8 mailbox registers |
| `ulPciLocDoorBell` | Pci to local doorbell reg |
| `ulLocPciDoorBell` | local to pci doorbell reg |
| `ulIntCntrlStat` | interrupt control/status |
| `ulRunTimeCntrl` | eeprom control, pci command codes, user I/O, init ctrl |
| `ulDeviceVendorID` | device id |
| `ulRevisionID` | revision id |
| `ulMailboxReg0` | mailbox register 0 |
| `ulMailboxReg1` | mailbox register 1 |
| `ulDMAMode0` | dma channel 0 mode |
| `ulDMAPCIAddress0` | dma channel 0 pci address |
| `ulDMALocalAddress0` | dma channel 0 local address |
| `ulDMAByteCount0` | dma channel 0 transfer byte count |
| `ulDMADescriptorPtr0` | dma channel 0 descriptor pointer |
| `ulDMAMode1` | dma channel 1 mode |
| `ulDMAPCIAddress1` | dma channel 1 pci address |
| `ulDMALocalAddress1` | dma channel 1 local address |
| `ulDMAByteCount1` | dma channel 1 transfer byte count |
| `ulDMADescriptorPtr1` | dma channel 1 descriptor pointer |
| `ulDMACmdStatus` | rw dma command/status registers |
| `ulDMAArbitration` | dma arbitration register |
| `ulDMAThreshold` | dma threshold register |
| `uchReserved3[12]` | Reserved |
| `ulMsgUnitCfg` | messaging unit configuration |
| `ulQBaseAddr` | queue base address register |
| `ulInFreeHeadPtr` | inbound free head pointer |
| `ulInFreeTailPtr` | inbound free tail pointer |
| `ulInPostHeadPtr` | inbound post head pointer |
| `ulInPostTailPtr` | inbound post tail pointer |
| `ulOutFreeHeadPtr` | outbound free head pointer |
| `ulOutFreeTailPtr` | outbound free tail pointer |
| `ulOutPostHeadPtr` | outbound post head pointer |
| `ulOutPostTailPtr` | outbound post tail pointer |
| `ulQStatusCtrl` | queue status/control |
| `uchReserved4[4]` | Reserved |
| `ulPciLocRange1` | range for pci to local 1 |
| `ulPciLocRemap1` | remap for pci to local 1 |
| `ulPciLBRegDescr1` | bus region descriptions for pci to local (space 1) |

## 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 24DSI32 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 “pci24dsi32_ioctl.h”

int 24DSI32_open(unsigned int board)
{
    int     fd;
    char    name[80];

    sprintf(name, "/dev/gsc24dsi32%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, 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 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 DMA_ORDER in the pci24dsi32.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.

### 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 "pci24dsi32_ioctl.h"

int 24DSI32_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 24DSI32 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 “pci24dsi32_ioctl.h”

int 24DSI32_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 24DSI32 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 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 "gsc24dsi32_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_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 a 24DSI32 register. This includes all PCI registers, all PLX PCI9080 feature set registers, and all GSC specific registers. Refer to `pci24dsi32_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 a 24DSI32 register. This includes only the GSC specific registers. All PCI and PLX PCI9080 feature set registers are read-only. Refer to `pci24dsi32_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_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_GSC_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_GSC_SET_SW_SYNCH

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

Usage

| `ioctl()` Argument | Description |
|---|---|
| `request` | `IOCTL_GSC_SET_SW_SYNCH` |

### 3.4.7. IOCTL_GSC_AUTO_CAL

Initiates an auto-calibration cycle.

Usage

| `ioctl()` Argument | Description |
|---|---|
| `request` | `IOCTL_GSC_AUTO_CAL` |

### 3.4.8. 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` |

### 3.4.9. IOCTL_GSC_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_GSC_SET_INITIATOR_MODE

Set this board as the initiator for synchronized acquisition. Enables the external synch output. Options are:

```
TARGET_MODE
INITIATOR_MODE
```

Usage

| **ioctl() Argument** | **Description** |
|---|---|
| request | IOCTL_GSC_SET_INITIATOR_MODE |
| arg | unsigned long * |

### 3.4.11. IOCTL_GSC_SET_BUFFER_THRESHOLD

Set the data buffer threshold register. Range is 0x0-0x3FFF (INPUT_BUFFER_SIZE).

Usage

| **ioctl() Argument** | **Description** |
|---|---|
| request | IOCTL_GSC_SET_BUFFER_THRESHOLD |
| arg | unsigned long * |

### 3.4.12. IOCTL_GSC_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_GSC_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_GSC_SET_GEN_RATE

This call provides information about the rate at which a specified generator should be set. The IOCTL_GSC_SET_GEN_RATE IOCTL command uses the GEN_RATE_PARAMS structure to set a generator's rate. 'eGenerator' specifies the generator, range being 0-3, and 'ulNrate' specifies the rate, range being 0x0-0x1FF.

Possible values for eGenerator are:

```
/* ---- generator codes ---- */
GEN_A
GEN_B
GEN_C
GEN_D
```

The following values are defined to assist computing a value for ulNrate:

```
/* ---- generator rate value limits ---- */
#define MIN_NRATE                           0x0
#define MAX_NRATE                           0x1FF
```

Usage

| `ioctl()` Argument | Description |
|---|---|
| `request` | `IOCTL_GSC_SET_GEN_RATE` |
| `arg` | `gen_rate_params *` |

### 3.4.15. IOCTL_GSC_ASSIGN_GEN_TO_GROUP

This call is used to assign a generator to a group consisting of four channels. The IOCTL_GSC_ASSIGN_GEN_TO_GROUP IOCTL command uses the gen_assign_params structure to assign a channel group to a specified generator.

```
/* ---- channel group codes ---- */
GRP_0
GRP_1
GRP_2
GRP_3

/* ---- generator assignment codes ---- */
ASN_GEN_A
ASN_GEN_B
ASN_GEN_C
ASN_GEN_D
ASN_EXT_CLK
ASN_GEN_DIRECT_EXT_CLK
ASN_GEN_NONE
```

Usage

| **ioctl()** **Argument** | **Description** |
|---|---|
| request | IOCTL_GSC_ASSIGN_GEN_TO_GROUP |
| arg | gen_assign_params * |

### 3.4.16. IOCTL_SET_RATE_DIVISOR

This call is used to set the value that divides the assigned rate generator frequency for a specified group of channels. The rate_divisor_params structure is used to set the channel divisor. 'ulGroup' specifies the channel group , and 'ulDivisor' specifies the frequency divisor value. The divisor is calculated using the generator frequency and the sampling rate.

```
typedef struct rate_divisor_params {
     unsigned long ulGroup;
     unsigned long ulDivisor;
} RATE_DIVISOR_PARAMS, *PRATE_DIVISOR_PARAMS;


      /* ---- channel group codes ---- */
      GRP_0
      GRP_1
      GRP_2
      GRP_3

      /* ---- rate divisor value limits ---- */
      #define MIN_NDIV                      0
      #define MAX_NDIV                      25
```

Usage

| **ioctl()** **Argument** | **Description** |
|---|---|
| request | IOCTL_GSC_SET_RATE_DIVISOR |
| arg | Rate_divisor_params * |

### 3.4.17. IOCTL_GET_DEVICE_ERROR

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

```
     GSC_SUCCESS
     GSC_INVALID_PARAMETER
     GSC_INVALID_BUFFER_SIZE
     GSC_PIO_TIMEOUT
     GSC_DMA_TIMEOUT
     GSC_IOCTL_TIMEOUT
     GSC_OPERATION_CANCELLED
     GSC_RESOURCE_ALLOCATION_ERROR
     GSC_INVALID_REQUEST
     GSC_AUTOCAL_FAILED
```

### 3.4.18. IOCTL_GSC_READ_PCI_CONFIG

Read the PCI configuration registers.

Usage

| `ioctl()` Argument | Description |
|---|---|
| request | IOCTL_GSC_READ_PCI_CONFIG |
| arg | device_read_pci_config_param * |

### 3.4.19. IOCTL_GSC_READ_LOCAL_CONFIG

Read the PLX configuration registers.

Usage

| `ioctl()` Argument | Description |
|---|---|
| request | IOCTL_GSC_READ_LOCAL_CONFIG |
| arg | device_register_params * |

### 3.4.20. IOCTL_GSC_SET_TIMEOUT

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

Usage

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

### 3.4.21. IOCTL_GSC_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 * |

### 3.4.22. IOCTL_GSC_GET_DEVICE_TYPE

Returns a unique enumerated type for each board type supported. Allows the use of multiple varieties of General Standards boards in the same system. See pci24dsi32_ioctl.h for a listing of board types supported by this driver.

Usage

| `ioctl()` Argument | Description |
|---|---|
| request | IOCTL_GSC_GET_DEVICE_TYPE |
| arg | unsigned long * |

### 3.4.23. IOCTL_GSC_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 *` |

### 3.4.24. IOCTL_GSC_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 *` |

### 3.4.25. IOCTL_GSC_SET_RATE_A_EXT_CLK

This IOCTL is used to set the state of the RATE-A external clock. When TRUE, selects the RATE-A generator as the external clock output source for the initiator mode. If low, selects the channel-00 sample clock.

Usage

| `ioctl()` Argument | Description |
|---|---|
| `Request` | `IOCTL_GSC_SET_RATE_A_EXT_CLK` |
| `Arg` | `unsigned long *` |

### 3.4.26. IOCTL_GSC_CLEAR_BUFFER_SYNC

This IOCTL is used to set the state of the software sync control bit. When TRUE, the software sync control bit becomes “clear buffer”. Default is FALSE.

Usage

| `ioctl()` Argument | Description |
|---|---|
| `Request` | `IOCTL_GSC_CLEAR_BUFFER_SYNC` |
| `Arg` | `unsigned long *` |

### 3.4.27. IOCTL_GSC_SET_OVERFLOW_LEVEL

This IOCTL is used to set the threshold level for the driver to check for the input buffer overfilling. The value passed is the actual trigger level, not, say, the number of words until full. The hardware does not support buffer overflow checking, so the driver checks to see if this buffer level has been exceeded when a read call is made. If it has, and checking is enabled, the driver fails the read and sets the global error code.

Usage

| `ioctl()` **Argument** | **Description** |
|---|---|
| `Request` | `IOCTL_GSC_SET_OVERFLOW_LEVEL` |
| `Arg` | `unsigned long *` |

### 3.4.28. IOCTL_GSC_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 *` |

### 3.4.29. IOCTL_GSC_SET_RATE_A_CLK_OUT

This IOCTL is used to set the source of the external clock. Possible values are:

```
EXT_CLK_RATE_A
EXT_CLK_GROUP_0
```

Usage

| `ioctl()` **Argument** | **Description** |
|---|---|
| `Request` | `IOCTL_GSC_SET_RATE_A_CLK_OUT` |
| `Arg` | `unsigned long *` |

### 3.4.30. IOCTL_GSC_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 *` |

### 3.4.31. IOCTL_GSC_SELECT_TTL_EXTERN_SYNC

This IOCTL is used to select differential LVDS or single-ended TTL external sync signals. The default is LVDS. Possible values are:

```
EXTERN_SYNC_TTL
EXTERN_SYNC_LVDS
```

Usage

| **`ioctl()` Argument** | **Description** |
|---|---|
| `Request` | `IOCTL_GSC_SELECT_TTL_EXTERN_SYNC` |
| `Arg` | `unsigned long *` |

### 3.4.32. IOCTL_GSC_SET_DATA_WIDTH

This IOCTL is used to select the output data width. 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 *` |

# 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 24DSI32. This is simply to address a few issues relating to using the 24DSI32.

## 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 24DSI32, 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 24DSI32 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 24DSI32 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 pci24dsi32.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.

## 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. |