SIO4

Four Channel High Speed Serial I/O

CPCI-SIO4/A/B
PCI-SIO4/A/B/BX
PMC-SIO4/A/B/BX
PC104P-SIO4B

Linux Device Driver User Manual

Manual Revision: September 30, 2005

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

Preface

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

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

Table of Contents

Table of Figures

1. Introduction

This user manual applies to driver version 1.18, release 2. This release of the driver is intended for those SIO4 models that use the Zilog Z16C30 dual USC chips.

> **WARNING:** Driver versions 1.06 and 1.18 and later are not run-time compatible with previous versions of the driver. Applications written to versions 1.05 and 1.17 and earlier must be rebuilt before using a later version. Additionally, various components of the 1.05 and earlier interface have been updated and a few are no longer available. Application may need a porting effort or they may optionally include the header `sio4_legacy.h` to continue using some of the available, but deprecated components. Some deprecated components are not available via this legacy header.
>
> **NOTE:** The device models listed on the front cover are those that are specifically supported by this release of the driver. Other models may be supported, though the level of support may vary. The driver may work with other SIO4 models, but performance may be degraded due to device feature and implementation differences.

1.1. Purpose

The purpose of this document is to describe the interface to the SIO4 Linux Device Driver. This software provides the interface between “Application Software” and the SIO4 board. The interface to this board is at the device level.

1.2. Acronyms

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

Acronyms	Description
DMA	Direct Memory Access
DMDMA	Demand Mode DMA
DPLL	Digital Phase Lock Loop
GSC	General Standards Corporation
PCI	Peripheral Component Interconnect
PMC	PCI Mezzanine Card
USC	Universal Serial Controller

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 SIO4 driver software executes under control of the Linux operating system and runs in Kernel Mode as a Kernel Mode device driver. The SIO4 device driver is implemented as a standard loadable Linux device driver written in the ‘C’ programming language. With the driver, user applications are able to open and close a channel and, while open, perform read, write and I/O control operations.

1.5. Hardware Overview

The SIO4 is a four channel high-speed serial interface I/O board. This board provides for bi-directional serial data transfers between two computers, or one computer and an external peripheral.

This board also can transfer data indefinitely without host intervention. Once the data link between the two computers is established, the desired transfers can be performed and will become transparent to the user.

The SIO4 board includes a DMA controller and comes with a maximum of 256K Bytes of FIFO storage, which is 32K per channel side (32K * 2 * 4). The FIFO configuration can vary greatly from one SIO4 version to another (i.e. 32K * 2 * 4 to 1K * 2 *1 to none at all). The SIO4 comes in an RS232 version and an RS485/422 version. Both versions include two Universal Serial Controllers (Zilog Z16C30 USC). The DMA controller is capable of transferring data to and from host memory; whereas the FIFO memory provides a means for continuous transfer of data without interrupting the DMA transfers or requiring intervention from the host CPU. The board also provides for interrupt generation for various states of the board like Sync Character detection, FIFO empty, FIFO full and DMA complete.

1.6. Reference Material

The following reference material may be of particular benefit in using the SIO4 and this driver. The specifications provide the information necessary for an in depth understanding of the specialized features implemented on this board.

- The applicable *SIO4 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/

- The *Z16C30 USC User's Manual* from Zilog. *

- The *Z16C30 Electronic Programmer's Manual* from Zilog (Zilog part number ZEPMDC00001). *

* The Zilog material is available from:

  Zilog, Inc.
  910 E Hamilton Ave
  CAMPBELL, CA 95008 USA
  Phone: 1-408-558-8500
  WEB: http://www.zilog.com/

2. Installation

2.1. CPU and Kernel Support

The driver is designed to operate with Linux kernel versions 2.2, 2.4 and 2.6 running on a PC system with one or more Intel x86 processors. Testing was performed under kernel versions 2.2.14 (Red Hat Linux 6.2), 2.4.21 (Red Hat Enterprise Linux Workstation Release 3) and 2.6.9 (Red Hat Enterprise Linux Workstation Release 4).

> **NOTE:** 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.
>
> **NOTE:** The driver has not been tested with a non-versioned kernel.
>
> **NOTE:** The driver has not been tested on an SMP host.

2.2. Compiler Support

The build for this driver relies on the use of the GCC compiler. This dependence is due only to the driver's use of the file `divdi3.c`, which is copied from GCC 2.95.1. The driver build process has been verified according to the above CPU and kernel support paragraph. The build process may fail under other build environments.

> **NOTE:** The dependence on the GCC compiler is due to the driver's use of 64-bit integer division. This division is performed during configuration of the programmable oscillator present on some versions of the SIO4. Under the 2.2 and 2.4 kernels the needed library services are linked implicitly during the build process. Under the 2.6 kernel build process, the needed services must be linked explicitly.

2.3. The /proc File System

While the driver is installed, the `/proc/sio4` file 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 then the entry value. Below is an example of what appears in the file, followed by descriptions of each entry.

```
version: 1.18
built: Sep 30 2005, 12:46:43
boards: 1
z16c30: 1
sync: 0
```

Entry	**Description**
`version:`	This gives the driver version number in the form `x.xx`.
`built:`	This gives the driver build date and time as a string. It is given in the C form of `printf("%s, %s", __DATE__, __TIME__).`
`boards:`	This identifies the total number of SIO4 boards the driver detected.
`z16c30:`	This identifies which installed boards are Z16C30 models, which use the Zilog Z16C30. A "`1`" indicates yes. A "`0`" indicates no. The order corresponds to the device node indexes in the `/dev` directory.
`sync:`	This identifies which installed boards are SYNC models. A "`1`" indicates yes. A "`0`" indicates no. The order corresponds to the device node indexes in the `/dev` directory.

2.4. File List

This release consists of the below listed files. The archive is described in detail in following subsections.

File	Description
`sio4.tar.gz`	This archive contains the entire driver with all associated files, including sources and make files.
`sio4_linux_driver_user_manual.pdf`	This is a PDF version of this user manual.

NOTE: Driver versions 1.05 and earlier were shipped as multiple archive files that were decompressed separately. As of version 1.06 all driver files (except the user manual) are supplied as a single archive. All previous files, archives and directories should be removed before proceeding with installation of this driver.

2.5. Directory Structure

The following table describes the directory structure observed by the source archive.

NOTE: As of version 1.06, the driver and all associated support files are installed under a single directory. All previous releases of the driver utilized a different directory structure based on where the user installed the separate archives. All previous files, archives and directories should be removed before proceeding with installation of this driver.

Directory Structure	Content
`sio4`	This is the driver's root directory.
`sio4/app3`	This directory contains the `sio4app3` sample application.
`sio4/app4`	This directory contains the `sio4app4` sample application.
`sio4/asyncc2c`	This directory contains the `asyncc2c` sample application.
`sio4/docsrc`	This directory contains the C sources from this user manual.
`sio4/driver`	This directory contains the driver executable and its sources.
`sio4/test`	This directory contains the `test` sample application.
`sio4/testloop`	This directory contains the `testloop` sample application.
`sio4/utils`	This directory contains utility sources used by the sample applications.

2.6. Installation

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

1. Change the current directory to `/usr/src/linux/drivers`.

   NOTE: Depending on the version of the kernel being used and the distribution, the `.../linux/...` portion of the directory may exist as a soft link or it may not. If it doesn't exist, then it should be present with a suffix including the base version number of the kernel. The directory should then appear as something like `.../linux-2.4/...`, or something similar. Perform the substitution as appropriate or create a `.../linux` soft link.

2. Copy the archive file `sio4.tar.gz` into the current directory.

3. Issue the following command to decompress and extract the files from the provided archive. This creates the directory structure described earlier and copies all of the archive files into the created directories.

   ```
   tar –xzvf sio4.tar.gz
   ```

2.7. The Driver

The paragraphs that follow give instructions on building, starting and verifying startup of the driver. These files are installed into the `/usr/src/linux/drivers/sio4/driver/` directory.

File	Description
`*.c`	These sources implement the driver interface and its functionality. Some functionality has been modularized based on individual source file base names.
`*.h`	These are driver header files. Others are listed below.
`makefile`	This is the driver make file.
`makefile.dep`	This is a make dependency file. This is updated automatically.
`sio4.h`	This is the driver interface header file. It should be included by SIO4 applications.
`sio4.o`	This is a pre-built version of the executable driver module.
`sio4_legacy.h`	This is an additional driver interface header file. This includes some outdated and superceded definitions. This should be included by applications on an as-needed basis.
`sio4_start`	This is a shell script to install the driver module and device nodes.

2.7.1. Build

Follow the below steps to build the driver.

1. Change to the directory where the driver and its sources were installed. This should be `/usr/src/linux/drivers/sio4/driver`.

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

   ```
   make clean
   ```

3. Build the driver by issuing the below command.

   ```
   make all
   ```

NOTE: Building the SIO4 driver requires installation of the kernel header sources. If they are not present the build will fail.

NOTE: Due to the differences between the many Linux distributions some build errors may occur. These errors may include system header location differences and should be easily correctable. Other errors may also appear as some distributors port newer kernel changes into older kernel distributions.

NOTE: Debug messages already present in the driver sources are enabled only when the macro `SIO4_DEBUG` is enabled. This macro is undefined by default and can be found in `sio4.h`.

2.7.2. Startup

NOTE: The driver may have to be rebuilt before being used due to kernel version differences between the GSC build host and the customer target host.

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

2.7.2.1. 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. This should be `/usr/src/linux/drivers/sio4/driver/`.

3. Install the driver module and create the device nodes by executing the below command. If any errors are encountered then an appropriate error message will be displayed.

   ```
   ./sio4_start
   ```

NOTE: The script's default specifies that the driver is installed in the same directory as the script. The script will fail if this is not so.

NOTE: The above step must be repeated each time the host is rebooted.

NOTE: The SIO4 device node major number is assigned dynamically by the kernel. The minor numbers and the device node suffix numbers are index numbers beginning with one, and increase by one for each additional serial channel.

4. Verify that the device module has been loaded by issuing the below command and examining the output. The module name `sio4` 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 four nodes for each installed board.

   ```
   ls –l /dev/sio4*
   ```

2.7.2.2. 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/sio4/driver/sio4_start
   ```

NOTE: The script's default specifies that the driver is installed in the same directory as the script. The startup script will fail if this is not so.

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 by following the verification steps given in the manual startup procedures.

2.7.3. Verification

Follow the below steps to verify that the driver has been properly installed and started.

1. Issue the below command to view the content of the driver's `/proc` file system text file.

```
cat /proc/sio4
```

2. If the file exists then the driver is installed and running.

2.7.4. Version

The driver version number can be obtained in a variety of ways. It is reported by the driver both when the driver is loaded and when it is unloaded (depending on kernel configuration options, this may be visible only in `/var/log/messages`). It is recorded in the file `/proc/sio4`. It can also be read by an application via the `SIO4_GET_DRIVER_INFO` IOCTL services.

2.7.5. 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 sio4
```

3. Verify that the driver module has been unloaded by issuing the below command. The module name `sio4` should not be in the list.

```
lsmod
```

2.8. Document Source Code Examples

The driver archive contains all of the source code examples included in this document. They are provided in library form appropriate for linking with SIO4 command line applications. The files are delivered undocumented and unsupported but may be used to assist in application development and to help ease the learning curve. The files are described here only briefly, though library use is described in the following paragraph. The purpose of the files is to provide the user with the actual source code given in the manual, to provide example code that can be used in customer applications, and to provide a means of insuring that the examples cited will compile.

2.8.1. SIO4DocSrcLib.a

This library comprises all code samples in this document along with some that are no longer in use. The files are installed into the `/usr/src/linux/drivers/sio4/docsrc/` directory. The files included are listed below.

NOTE: This source code library is common across all SIO4 models supported by the driver and may include some code samples intended for SIO4 models not listed on the cover page of this user manual. The sample code may work with other unlisted models, but may not function as expected since they may not necessarily be intended for these models. Library code not referenced in this user manual should generally not be used with those models listed on the cover page. For other SIO4 models, refer to the applicable driver user manual.

File	Description
`*.c`	These are the sources which correspond to the samples in this user manual.
`makefile`	This is the library make file.
`makefile.dep`	This is a make dependency file. This is updated automatically.
`SIO4DocSrcLib.a`	This is a pre-built version of the library.
`SIO4DocSrcLib.h`	This is a header file that gives the prototypes for all library functions.

2.8.1.1. Build

Follow the below steps to compile the example files.

1. Change to the directory where the source code example files were installed. This should be `/usr/src/linux/drivers/sio4/docsrc/`.

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

   ```
   make clean
   ```

3. Compile the sample files by issuing the below command.

   ```
   make all
   ```

2.8.1.2. Use

Use of the library has requirements applicable at the time the application is being built. C sources using the library must include the header file, which is named `SIO4DocSrcLib.h`. Applications using the library must link the file `SIO4DocSrcLib.a`. Both the include search path and the library search path must be expanded to include the directory `/usr/src/linux/drivers/sio4/docsrc/`.

2.9. Sample Applications

The driver archive contains three sample applications. These are Linux user mode application programs. They are delivered undocumented and unsupported but may be used to exercise the SIO4 and its device driver. They can also be used as starting points for developing applications on top of the Linux driver and to help ease the learning curve. The applications are described in the following paragraphs.

> **NOTE:** These sample applications are designed to function with the SIO4 models listed on the cover of this user manual. The sample applications may work with other models, but may not function as expected since they are not necessarily intended for those models. Refer to the driver user manual and sample applications supplied with the SIO4 model in question, as applicable.

2.9.1. sio4Test.o

This sample application provides a menu driven Linux application that can be used to test the different features that are supported by the SIO4 device driver. It can be used as the starting point for any application development on top of the SIO4 Linux device driver. The menu provided with the application is generally self-explanatory. The files are installed into the `/usr/src/linux/drivers/sio4/test/` directory. This application includes the below listed files.

> **NOTE:** This application was designed for driver version 1.05 and earlier. It has not been ported to the current driver and may be removed from future driver releases. Since it has not been ported some of the options may not function properly..

> **NOTE:** Before doing any operation on a channel, the channel must be selected and opened using the menu selections A and B.

File	**Description**
`makeapp.sh`	This is the build script for this sample application.
`sio4Test.c`	This is the source file.
`sio4Test.o`	This is the pre-built test application.
`sio4test.h`	A header for this test application.

2.9.1.1. Build

Follow the below steps to build/rebuild the sample application.

1. Change to the directory where the sample application was installed. This should be `/usr/src/linux/drivers/sio4/test/`.

2. Build the sample application by issuing the below command.

   ```
   ./makeapp.sh
   ```

2.9.1.2. Execute

Follow the below steps to execute and exercise the test application.

1. Change to the directory where the sample application was installed. This should be `/usr/src/linux/drivers/sio4/test/`.

2. Start the test application by issuing the command given below.

   ```
   ./sio4Test.o
   ```

3. The application presents a menu of options, which should generally be self explanatory. To exercise the application, and consequentially the driver and the installed board, select from the menu items presented. When finished, select the exit option.

2.9.2. sio4LoopTest.o

This sample application performs loop back tests. The purpose of these tests is to verify the operation of different portions of the SIO4 board's circuitry. The menu also has an option to perform all the tests on all channels without user intervention, except for the cable loop mode, which has to be run separately. The menu provided with the application is generally self-explanatory and the application logs the test results in a log file named `sio4log`, which is created in the current directory. The files are installed into the directory `/usr/src/linux/drivers/sio4/testloop/`. The source files included and the specific tests performed are described below.

NOTE: This application was designed for driver version 1.05 and earlier. It has not been ported to the current driver and may be removed from future driver releases.

File	Description
`async.h`	A header for this test application.
`makeapp.sh`	This is the build script for this sample application.
`sio4LoopTest.c`	This is the source file.
`sio4LoopTest.o`	This is the pre-built test application.
`sio4test.h`	A header file for this test application.

2.9.2.1. Build

Follow the below steps to build/rebuild the sample application.

1. Change to the directory where the sample application was installed. This should be `/usr/src/linux/drivers/sio4/testloop/`.

2. Build the sample application by issuing the below command.

```
./makeapp.sh
```

2.9.2.2. Execute

Follow the below steps to execute and exercise the test application.

1. Change to the directory where the sample application was installed. This should be `/usr/src/linux/drivers/sio4/testloop/`.

2. Start the test application by issuing the command given below.

   ```
   ./sio4LoopTest.o
   ```

3. The application presents a menu of options, which should generally be self explanatory. To exercise the application, and consequentially the driver and the installed board, select from the menu items presented. When finished, select the exit option.

2.9.2.3. Internal Local Loop

In this test the Z16C30 is put into its INTERNAL LOCAL LOOP mode and the cable signals are all disabled. In this test, data is routed via the Z16C30's INTERNAL LOCAL LOOP test mode circuitry. Since the SIO4 cable signals are disabled for this test, the cable type in use, if any, is ignored.

2.9.2.4. Local Loop

In this test the Z16C30 is put into its LOCAL LOOP mode and the cable signals are all disabled. In this test, data is routed via the Z16C30's LOCAL LOOP test mode circuitry. Since the SIO4 cable signals are disabled for this test, the cable type in use, if any, is ignored.

2.9.2.5. External Loop High

In this test the Z16C30 is put into its NORMAL mode and the cable signals are configured so the transmit and receive channels are both connected to the cable's HIGH side. In this test, data is routed via the SIO4 board's HIGH side transceivers. Since some SIO4 cable signals are enabled, the cable type in use, if any, must be of a known type (i.e. either no cable at all or each channel's transmitter and receiver must be tied together).

2.9.2.6. External Loop Low

In this test the Z16C30 is put into its NORMAL mode and the cable signals are configured so the transmit and receive channels are both connected to the cable's LOW side. In this test, data is routed via the SIO4 board's LOW side transceivers. Since some SIO4 cable signals are enabled, the cable type in use, if any, must be of a known type (i.e. either no cable at all or each channel's transmitter and receiver must be tied together).

NOTE: This test is applicable only to RS485/422 version SIO4 boards.

2.9.2.7. Cable Loop

In this test the Z16C30 is put into its NORMAL mode and the cable signals are configured so that the transmitter is connected to the HIGH side and the receiver is connected to the LOW side. In this test, data is routed via an externally attached cable. For this test, the cable used must route each channel's transmit lines to its own receive lines.

NOTE: This test requires an external cable/connector that connects each channel’s transmitter signals to its own receiver signals.

2.9.3. sio4app3

This sample application performs an automated test of numerous, more recently added IOCTL service. It can be used as the starting point for any application development on top of the SIO4 Linux device driver. The files are installed into the directory `/usr/src/linux/drivers/sio4/app3/`. This application includes the below listed files.

File	Description
`*.c`	These are the sources which implement the IOCTL test code. Most contain functionality based on the file's base name.
`app3.h`	This is a header used by application.
`makefile`	This is the driver make file.
`makefile.dep`	This is a make dependency file. This is updated automatically.
`sio4app3`	This is a pre-built version of the application.

2.9.3.1. Build

Follow the below steps to build/rebuild the sample application.

1. Change to the directory where the sample application was installed. This should be `/usr/src/linux/drivers/sio4/app3/`.

2. Build the sample application by issuing the below command.

   ```
   make clean all
   ```

2.9.3.2. Execute

Follow the below steps to execute and exercise the test application.

1. Change to the directory where the sample application was installed. This should be `/usr/src/linux/drivers/sio4/app3/`.

2. Start the test application by issuing the command given below. The `-c` argument (lower case) requests continuous testing until a test error occurs. The `-C` argument (upper case) requests continuous testing without regard to test error. The `-m#` argument requests that continuous testing run for at most # minutes, where # is a decimal number. The `-n#` argument requests that continuous testing run for at most # tests, where # is a decimal number. The `index` argument is the zero based index of the board to be accessed. All arguments are optional.

   ```
   ./sio4app3 <-c> <-C> <-m#> <-n#> <index>
   ```

3. The application will report board identification information then perform a series of automated tests. Each test cycle should complete in less than one minute.

2.9.4. sio4app4

This sample application performs an automated test of a small number of IOCTL service. It can be used as the starting point for any application development on top of the SIO4 Linux device driver. The files are installed into the directory `/usr/src/linux/drivers/sio4/app4/`. This application includes the below listed files.

File	Description
`*.c`	These are the sources which implement the IOCTL test code. Most contain functionality based on the file's base name.

`main.h`	This is a header used by application.
`makefile`	This is the driver make file.
`makefile.dep`	This is a make dependency file. This is updated automatically.
`sio4app4`	This is a pre-built version of the application.

2.9.4.1. Build

Follow the below steps to build/rebuild the sample application.

1. Change to the directory where the sample application was installed. This should be `/usr/src/linux/drivers/sio4/app4/`.

2. Build the sample application by issuing the below command.

   ```
   make clean all
   ```

2.9.4.2. Execute

Follow the below steps to execute and exercise the test application.

1. Change to the directory where the sample application was installed. This should be `/usr/src/linux/drivers/sio4/app4/`.

2. Start the test application by issuing the command given below. The `-c` argument (lower case) requests continuous testing until a test error occurs. The `-C` argument (upper case) requests continuous testing without regard to test error. The `-lxxx` argument specifies the type of loop back cable that is attached. Use `-l232` if an RS232 loop back cable is attached, `-l485` if an RS485 loop back cable is attached, `-lnone` if a loop back cable is not attached, and `-lother` if some other cable type is attached. The `-m#` argument requests that continuous testing run for at most `#` minutes, where `#` is a decimal number. The `-n#` argument requests that continuous testing run for at most `#` tests, where `#` is a decimal number. The `index` argument is the zero based index of the board to be accessed. All arguments are optional.

   ```
   ./sio4app4 <-c> <-C> <-lxxx> <-m#> <-n#> <index>
   ```

3. The application will report board identification information then perform a series of automated tests. Each test cycle should complete in less than five minutes.

2.9.5. asyncc2c

This sample application performs asynchronous data transfer between two SIO4 channels specified on the command line. It can be used as the starting point for any application development on top of the SIO4 Linux device driver. The files are installed into the directory `/usr/src/linux/drivers/sio4/asyncc2c/`. This application includes the below listed files.

File	**Description**
`*.c`	These are the sources which implement the functionality specific to this application.
`async.h`	This is the configuration header defining all of the relevant parameter options.
`asyncc2c`	This is a pre-built version of the application.
`main.h`	This is the application’s common header file.
`makefile`	This is the driver make file.
`makefile.dep`	This is a make dependency file. This is updated automatically.

2.9.5.1. Build

Follow the below steps to build/rebuild the sample application.

1. Change to the directory where the sample application was installed. This should be `/usr/src/linux/drivers/sio4/asyncc2c/`.

2. Build the sample application by issuing the below command.

   ```
   make clean all
   ```

2.9.5.2. Execute

Follow the below steps to execute and exercise the test application.

1. Change to the directory where the sample application was installed. This should be `/usr/src/linux/drivers/sio4/asyncc2c/`.

2. Start the test application by issuing the command given below. The `-c` argument (lower case) requests continuous testing until a test error occurs. The `-C` argument (upper case) requests continuous testing without regard to test error. The `-m#` argument requests that continuous testing run for at most `#` minutes, where `#` is a decimal number. The `-n#` argument requests that continuous testing run for at most `#` tests, where `#` is a decimal number. The `index1` and `index2` arguments are the zero based indexes of the two channels to be tested. If the index numbers are identical, then an internal loop back mode is used. Otherwise a cable is required for data transfer between the two specified channels. The index arguments are required. All others are optional.

   ```
   ./asyncc2c <-c> <-C> <-m#> <-n#> <index1> <index2>
   ```

3. The application will report board identification information then perform appropriate channel-to-channel data transfers. Each individual data transfer test will require a minimum of about 10 seconds. The maximum amount of time required is generally the amount of time required to transfer two FIFOs worth of data in both directions at the specified baud rate. Overall, test iterations should take less than five minutes.

2.10. Removal

Follow the below steps to remove the driver and all associated files.

1. Shutdown the driver as described in 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 sio4.tar.gz sio4
   ```

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

   ```
   rm -f /dev/sio4*
   ```

5. If the automated startup procedure was adopted, then edit the system startup script `rc.local` and remove the line that invokes the `sio4_start` script. The file `rc.local` should be located in the `/etc/rc.d` directory.

3. Driver Interface

The SIO4 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 standard driver interface to the GSC SIO4 board for Linux applications. The interface includes various macros, data types and functions, all of which are described in the following paragraphs. The SIO4 specific portion of the driver interface is defined in the header file `sio4.h`, portions of which are described in this section. The header defines numerous items in addition to those described here. Some definitions from previous versions of the interface are available as needed by also including the header file `sio4_legacy.h`.

NOTE: Contact General Standards Corporation if additional driver functionality is required.

WARNING: Driver versions 1.06 and 1.18 and later are not run-time compatible with previous versions of the driver. Applications written to versions 1.05 and 1.17 and earlier must be rebuilt before using a later version. Additionally, various components of the 1.05 and earlier interface have been updated and a few are no longer available. Application may need a porting effort or they may optionally include the header `sio4_legacy.h` to continue using some of the available, but deprecated components. Some deprecated components are not available via this legacy header.

NOTE: The driver included with this release is designed to work with the SIO4 models listed on the cover page of this user manual, as well as other models not listed. The driver interface may therefore include IOCTL services and support components intended for use with other models. Services and support components not documented in this manual should therefore not be used with the models listed on the cover. For other SIO4 models, refer to the applicable driver user manual.

3.1. Macros

The driver interface includes the following macros which are defined in `sio4.h`. This header contains numerous additional utility type macros in addition to those described here.

3.1.1. FIFO Count Options

This set of macros gives the predefined options returned by the `SIO4_TX_FIFO_COUNT` and `SIO4_RX_FIFO_COUNT` IOCTL services. Other values returned reflect the FIFO's current fill level.

Macros	Description
`SIO4_FIFO_COUNT_UNKNOWN`	The FIFO fill level is unknown.

3.1.2. FIFO Size Options

This set of macros gives the predefined options returned by the `SIO4_TX_FIFO_SIZE` and `SIO4_RX_FIFO_SIZE` IOCTL services. Other values returned reflect the actual size of the respective FIFO's.

Macros	Description
`SIO4_FIFO_SIZE_UNKNOWN`	The FIFO size is unknown.

3.1.3. FIFO Type Options

This set of macros defines the options returned by the `SIO4_TX_FIFO_TYPE` and `SIO4_RX_FIFO_TYPE` IOCTL services.

Macros	Description
`SIO4_FIFO_TYPE_HARD`	The FIFO consists of hardware chips located on the board.
`SIO4_FIFO_TYPE_SOFT`	The FIFO consists of firmware logic inside the FPGA.
`SIO4_FIFO_TYPE_UNKNOWN`	The FIFO type is unknown.

3.1.4. IOCTL

The IOCTL macros are documented following the function call descriptions.

3.1.5. Registers

The following table gives the complete set of SIO4 registers. The tables are divided by register categories.

3.1.5.1. GSC Registers

The following table gives the complete set of GSC specific SIO4 registers. For detailed definitions of these registers refer to the *SIO4 User Manual*.

Macros	Description
`SIO4_GSC_BCR`	Board Control Register
`SIO4_GSC_BSR`	Board Status Register
`SIO4_GSC_CCR`	Clock Control Register
`SIO4_GSC_CSR`	Control/Status Register
`SIO4_GSC_FCR`	FIFO Count Register
`SIO4_GSC_FDR`	FIFO Data Register
`SIO4_GSC_FR`	Features Register
`SIO4_GSC_FRR`	Firmware Revision Register
`SIO4_GSC_FSR`	FIFO Size Register
`SIO4_GSC_ICR`	Interrupt Control Register
`SIO4_GSC_ISR`	Interrupt Status Register
`SIO4_GSC_IELR`	Interrupt Edge/Level Register
`SIO4_GSC_IHLR`	Interrupt Hi/low Register
`SIO4_GSC_POCSR`	Programmable Oscillator Control/Status Register
`SIO4_GSC_PORAR`	Programmable Oscillator RAM Address Register
`SIO4_GSC_PORDR`	Programmable Oscillator RAM Data Register
`SIO4_GSC_PSRCR`	Pin Source Register
`SIO4_GSC_RAR`	Receiver Almost Empty/Full Register
`SIO4_GSC_SBR`	Sync Byte Register
`SIO4_GSC_TAR`	Transmitter Almost Empty/Full Register

3.1.5.2. PCI Configuration Registers

The following table gives the complete set of PCI Configuration Registers. For detailed definitions of these registers refer to the *PCI9080 Data Book*. All PCI registers are accessed by their native data size. The exception is SIO4_PCI_CCR, which is 24-bits wide but accessed as if it were 32-bits. Since the interface permits retrieval only of 32-bit values, the insignificant upper bits can be ignored and should be zero.

PCI Configuration Registers

Macros	Description
`SIO4_PCI_BAR0`	PCI Base Address Register for Memory Accesses to Local, Runtime, and DMA Registers (PCIBAR0)
`SIO4_PCI_BAR1`	PCI Base Address Register for I/O Accesses to Local, Runtime, and DMA Registers (PCIBAR1)

`SIO4_PCI_BAR2`	PCI Base Address Register for Memory Accesses to Local Address Space 0 (PCIBAR2)
`SIO4_PCI_BAR3`	PCI Base Address Register for Memory Accesses to Local Address Space 1 (PCIBAR3)
`SIO4_PCI_BAR4`	Unused Base Address Register (PCIBAR4)
`SIO4_PCI_BAR5`	Unused Base Address Register (PCIBAR5)
`SIO4_PCI_BISTR`	PCI Built-In Self Test Register (PCIBISTR)
`SIO4_PCI_CCR`	PCI Class Code Register (PCICCR)
`SIO4_PCI_CIS`	PCI Cardbus CIS Pointer Register (PCICIS)
`SIO4_PCI_CLSR`	PCI Cache Line Size Register (PCICLSR)
`SIO4_PCI_CR`	PCI Command Register (PCICR)
`SIO4_PCI_DID`	PCI Device ID Register (PCIDIDR)
`SIO4_PCI_ERBAR`	PCI Expansion ROM Base Address (PCIERBAR)
`SIO4_PCI_HTR`	PCI Header Type Register (PCIHTR)
`SIO4_PCI_ILR`	PCI Interrupt Line Register (PCIILR)
`SIO4_PCI_IPR`	PCI Interrupt Pin Register (PCIIPR)
`SIO4_PCI_LTR`	PCI Latency Timer Register (PCILTR)
`SIO4_PCI_MGR`	PCI Min_Gnt Register (PCIMGR)
`SIO4_PCI_MLR`	PCI Max_Lat Register (PCIMLR)
`SIO4_PCI_REV`	PCI Revision ID Register (PCIREV)
`SIO4_PCI_SID`	PCI Subsystem ID Register (PCISID)
`SIO4_PCI_SR`	PCI Status Register (PCISR)
`SIO4_PCI_SVID`	PCI Subsystem Vendor ID Register (PCISVID)
`SIO4_PCI_VID`	PCI Vendor ID Register (PCIVIDR)

NOTE: A PCIVIDR value of `0x10B5` and a PCIDIDR value of `0x9080` identify the PCI interface chip as a PLX PCI9080. A PCISVID value of `0x10B5` identifies that the PCISID was assigned by PLX. A PCISID value of `0x2401` identifies the SIO4.

3.1.5.3. PLX Feature Set Registers

The following table gives the complete set of PLX Feature Set Registers, which are the feature specific registers implemented by the PLX PCI9080. The PLX PCI9080 is the PCI bridge chip used on the SIO4. For detailed definitions of these registers refer to the *PCI9080 Data Book*. All PLX registers are accessed by their native data size. Since the interface permits retrieval only of 32-bit values, the insignificant upper bits can be ignored and should be zero.

Local Configuration Registers

Macros	**Description**
`SIO4_PLX_BIGEND`	Big/Little Endian Descriptor Register (BIGEND)
`SIO4_PLX_EROMBA`	Expansion ROM Local Base Address Register (EROMBA)
`SIO4_PLX_EROMRR`	Expansion ROM Range Register (EROMRR)
`SIO4_PLX_DMCFGA`	PCI Configuration Address Register for Direct Master to PCI IO/CFG (DMCFGA)
`SIO4_PLX_DMLBAI`	Local Base Address Register for Direct Master to PCI IO/CFG (DMLBAI)
`SIO4_PLX_DMLBAM`	Local Base Address Register for Direct Master to PCI Memory (DMLBAM)
`SIO4_PLX_DMPBAM`	PCI Base Address Register for Direct Master to PCI Memory (DMPBAM)
`SIO4_PLX_DMRR`	Local Range Register for Direct Master to PCI (DMRR)
`SIO4_PLX_LAS0BA`	Local Address Space 0 Local Base Address Register (LAS0BA)
`SIO4_PLX_LAS0RR`	Local Address Space 0 Range Register for PCI-to-Local Bus (LAS0RR)
`SIO4_PLX_LAS1BA`	Local Address Space 1 Local Base Address Register (LAS1BA)
`SIO4_PLX_LAS1RR`	Local Address Space 1 Range Register for PCI-to-Local Bus (LAS1RR)
`SIO4_PLX_LBRD0`	Local Address Space 0/Expansion ROM Bus Region Descriptor Register (LBRD0)
`SIO4_PLX_LBRD1`	Local Address Space 1 Bus Region Descriptor Register (LBRD1)
`SIO4_PLX_MARBR`	Mode Arbitration Register (MARBR)

Runtime Registers

Macros	Description
SIO4_PLX_CNTRL	Serial EEPROM Control, CPI Command Codes, User I/O, Init Control Register (CNTRL)
SIO4_PLX_INTCSR	Interrupt Control/Status Register (INTCSR)
SIO4_PLX_L2PDBELL	Local-to-PCI Doorbell Register (L2PDBELL)
SIO4_PLX_MBOX0	Mailbox Register 0 (MBOX0)
SIO4_PLX_MBOX1	Mailbox Register 1 (MBOX1)
SIO4_PLX_MBOX2	Mailbox Register 2 (MBOX2)
SIO4_PLX_MBOX3	Mailbox Register 3 (MBOX3)
SIO4_PLX_MBOX4	Mailbox Register 4 (MBOX4)
SIO4_PLX_MBOX5	Mailbox Register 5 (MBOX5)
SIO4_PLX_MBOX6	Mailbox Register 6 (MBOX6)
SIO4_PLX_MBOX7	Mailbox Register 7 (MBOX7)
SIO4_PLX_P2LDBELL	PCI-to-Local Doorbell Register (P2LDBELL)
SIO4_PLX_PCIHIDR	PCI Permanent Configuration ID Register (PCIHIDR)
SIO4_PLX_PCIHREV	PCI Permanent Revision ID Register (PCIHREV)

DMA Registers

Macros	Description
SIO4_PLX_DMAARB	DMA Arbitration Register (DMAARB)
SIO4_PLX_DMACSR0	DMA Channel 0 Command/Status Register (DMACSR0)
SIO4_PLX_DMACSR1	DMA Channel 1 Command/Status Register (DMACSR1)
SIO4_PLX_DMADPR0	DMA Channel 0 Descriptor Pointer Register (DMADPR0)
SIO4_PLX_DMADPR1	DMA Channel 1 Descriptor Pointer Register (DMADPR1)
SIO4_PLX_DMALADR0	DMA Channel 0 Local Address Register (DMALADR0)
SIO4_PLX_DMALADR1	DMA Channel 1 Local Address Register (DMALADR1)
SIO4_PLX_DMAMODE0	DMA Channel 0 Mode Register (DMAMODE0)
SIO4_PLX_DMAMODE1	DMA Channel 1 Mode Register (DMAMODE1)
SIO4_PLX_DMAPADR0	DMA Channel 0 PCI Address Register (DMAPADR0)
SIO4_PLX_DMAPADR1	DMA Channel 1 PCI Address Register (DMAPADR1)
SIO4_PLX_DMASIZ0	DMA Channel 0 Transfer Size Register (DMASIZ0)
SIO4_PLX_DMASIZ1	DMA Channel 1 Transfer Size Register (DMASIZ1)
SIO4_PLX_DMATHR	DMA Threshold Register (DMATHR)

Message Queue Registers

Macros	Description
SIO4_PLX_IFHPR	Inbound Free Head Pointer Register (IFHPR)
SIO4_PLX_IFTPR	Inbound Free Tail Pointer Register (IFTPR)
SIO4_PLX_IPHPR	Inbound Post Head Pointer Register (IPHPR)
SIO4_PLX_IPTPR	Inbound Post Tail Pointer Register (IPTPR)
SIO4_PLX_IQP	Inbound Queue Port Register (IQP)
SIO4_PLX_MQCR	Messaging Queue Configuration Register (MQCR)
SIO4_PLX_OFHPR	Outbound Free Head Pointer Register (OFHPR)
SIO4_PLX_OFTPR	Outbound Free Tail Pointer Register (OFTPR)
SIO4_PLX_OPHPR	Outbound Post Head Pointer Register (OPHPR)
SIO4_PLX_OPLFIM	Outbound Post List FIFO Interrupt Mask Register (OPLFIM)
SIO4_PLX_OPLFIS	Outbound Post List FIFO Interrupt Status Register (OPLFIS)
SIO4_PLX_OPTPR	Outbound Post Tail Pointer Register (OPTPR)
SIO4_PLX_OQP	Outbound Queue Port Register (OQP)

SIO4_PLX_QBAR	Queue Base Address Register (QBAR)
SIO4_PLX_QSR	Queue Status/Control Register (QSR)

3.1.5.4. Zilog USC Registers

The following table gives the complete set of Zilog USC registers.

Macros	Description
SIO4_USC_CCAR	Channel Command/Address Register (CCAR)
SIO4_USC_CCR	Channel Control Register (CCR)
SIO4_USC_CCSR	Channel Command/Status Register (CCSR)
SIO4_USC_CMCR	Clock Mode Control Register (CMCR)
SIO4_USC_CMR	Channel Mode Register (CMR)
SIO4_USC_DCCR	Daisy Chain Control Register (DCCR)
SIO4_USC_HCR	Hardware Configuration Register (HCR)
SIO4_USC_ICR	Interrupt Control Register (ICR)
SIO4_USC_IOCR	Input/Output Control Register (IOCR)
SIO4_USC_IVR	Interrupt Vector Register (IVR)
SIO4_USC_MISR	Miscellaneous Interrupt Status Register (MISR)
SIO4_USC_RCCR	Receive Character Count Register (RCCR)
SIO4_USC_RCLR	Receive Count Limit Register (RCLR)
SIO4_USC_RCSR	Receive Command/Status Register (RCSR)
SIO4_USC_RDR	Receive Data Register (RDR)
SIO4_USC_RICR	Receive Interrupt Control Register (RICR)
SIO4_USC_RMR	Receive Mode Register (RMR)
SIO4_USC_RSR	Receive Sync Register (RSR)
SIO4_USC_SICR	Status Interrupt Control Register (SICR)
SIO4_USC_TC0R	Time Constant 0 Register (TC0R)
SIO4_USC_TC1R	Time Constant 1 Register (TC1R)
SIO4_USC_TCCR	Transmit Character Count Register (TCCR)
SIO4_USC_TCLR	Transmit Count Limit Register (TCLR)
SIO4_USC_TCSR	Transmit Command/Status Register (TCSR)
SIO4_USC_TDR	Transmit Data Register (TDR)
SIO4_USC_TICR	Transmit Interrupt Control Register (TICR)
SIO4_USC_TMCR	Test Mode Control Register (TMCR)
SIO4_USC_TMDR	Test Mode Data Register (TMDR)
SIO4_USC_TMR	Transmit Mode Register (TMR)
SIO4_USC_TSR	Transmit Sync Register (TSR)

3.1.6. SIO4_BOARD_JUMPERS Options

This set of macros gives the predefined options available for the SIO4_BOARD_JUMPERS IOCTL service.

Macros	Description
SIO4_BOARD_JUMPERS_UNSUPPORTED	The board jumpers are unsupported on the reference board.

3.1.7. SIO4_ CABLE_CONFIG Options

This set of macros defines the options available for the SIO4_CABLE_CONFIG IOCTL service (see page 53). The signals referred to in the table are the data and clock signals for the transmitter and the receiver. Also, "lower" refers to the cable pins designated as "lower pins", and "upper" refers to the cable pins designated as "upper pins" Refer to the header sio4.h for additional utility macros.

Macros (set and get options)	**Description**
`SIO4_CABLE_CONFIG_TXDIS_RXDIS`	The Tx and Rx signals are disabled.
`SIO4_CABLE_CONFIG_TXDIS_RXLWR`	The Tx signals are disabled and the Rx lower signals are enabled.
`SIO4_CABLE_CONFIG_TXDIS_RXUPR`	The Tx signals are disabled and the Rx upper signals are enabled.
`SIO4_CABLE_CONFIG_TXLWR_RXDIS`	The Tx lower signals are enabled and the Rx signals are disabled.
`SIO4_CABLE_CONFIG_TXLWR_RXUPR`	The Tx lower signals are enabled and the Rx upper signals are enabled.
`SIO4_CABLE_CONFIG_TXUPR_RXDIS`	The Tx upper signals are enabled and the Rx signals are disabled.
`SIO4_CABLE_CONFIG_TXUPR_RXLWR`	The Tx upper signals are enabled and the Rx lower signals are enabled.

Macros (set only options)	**Description**
`SIO4_CABLE_CONFIG_READ`	Retrieve the current configuration.

Macros (get only options)	**Description**
`SIO4_CABLE_CONFIG_INVALID`	The current configuration is invalid.
`SIO4_CABLE_CONFIG_UNKNOWN`	The current configuration is unknown.

3.1.8. SIO4_CLEAR_DPLL_STATUS Options

This set of macros defines the bits that may be combined to form the value passed to the `SIO4_CLEAR_DPLL_STATUS` IOCTL service and refers to DPLL status bits in the USC CCSR.

Macros	**Description**
`CLEAR_DPLL_ALL_STATUS`	Clear all three of the DPLL status bits.
`CLEAR_DPLL_IN_SYNC`	Clear the DPLL Sync bit.
`CLEAR_DPLL_MISSING_1_CLOCK`	Clear the DPLL1Miss bit.
`CLEAR_DPLL_MISSING_2_CLOCKS`	Clear the DPLL2Miss bit.

3.1.9. SIO4_CTS_CABLE_CONFIG Options

This set of macros defines the options available for the `SIO4_CTS_CABLE_CONFIG` IOCTL service (see page 54).

Macros (set and get options)	**Description**
`SIO4_CTS_CABLE_CONFIG_CTS_IN`	The cable CTS signal is the CTS input to the USC.
`SIO4_CTS_CABLE_CONFIG_DCD_IN`	The cable DCD signal is the CTS input to the USC.
`SIO4_CTS_CABLE_CONFIG_DISABLE`	Disable use of the CTS signal.
`SIO4_CTS_CABLE_CONFIG_DRV_LOW`	An output driven low.
`SIO4_CTS_CABLE_CONFIG_DRV_HI`	An output driven high.
`SIO4_CTS_CABLE_CONFIG_RTS_OUT`	The cable CTS signal is an RTS output, which is the FIFO Full status from the Rx FIFO.

Macros (set only options)	**Description**
`SIO4_CTS_CABLE_CONFIG_READ`	Retrieve the current configuration.

Macros (get only options)	**Description**
`SIO4_CTS_CABLE_CONFIG_INVALID`	The current configuration is invalid.
`SIO4_CTS_CABLE_CONFIG_UNKNOWN`	The current configuration is unknown.

3.1.10. SIO4_DCD_CABLE_CONFIG Options

This set of macros defines the options available for the `SIO4_DCD_CABLE_CONFIG` IOCTL service (see page 55).

Macros (set and get options)	Description
SIO4_DCD_CABLE_CONFIG_CTS_IN	The cable CTS signal is the DCD input to the USC. The USC uses the signal for Data Carrier Detect operation.
SIO4_DCD_CABLE_CONFIG_CTS_IN_SYNC	The cable CTS signal is the DCD input to the USC. The USC uses the signal for Data Sync Detect operation.
SIO4_DCD_CABLE_CONFIG_DCD_IN	The cable DCD signal is the DCD input to the USC. The USC uses the signal for Data Carrier Detect operation.
SIO4_DCD_CABLE_CONFIG_DCD_IN_SYNC	The cable DCD signal is the DCD input to the USC. The USC uses the signal for Data Sync Detect operation.
SIO4_DCD_CABLE_CONFIG_DISABLE	Disable use of the DCD signal.
SIO4_DCD_CABLE_CONFIG_DRV_HI	An output driven high.
SIO4_DCD_CABLE_CONFIG_DRV_LOW	An output driven low.
SIO4_DCD_CABLE_CONFIG_RTS_OUT	The cable DCD signal is an RTS output, which is the FIFO Full status from the Rx FIFO.

Macros (set only options)	Description
SIO4_DCD_CABLE_CONFIG_READ	Retrieve the current configuration.

Macros (get only options)	Description
SIO4_DCD_CABLE_CONFIG_INVALID	The current configuration is invalid.
SIO4_DCD_CABLE_CONFIG_UNKNOWN	The current configuration is unknown.

3.1.11. SIO4_FEATURE_TEST Options

This set of macros defines the options available for the SIO4_FEATURE_TEST IOCTL service.

Macros (features/set only options)	Description
SIO4_FEATURE_BCR_BOARD_RESET	Does the Board Control Register support the Board Reset bit?
SIO4_FEATURE_BCR_RX_FFC	Does the Board Control Register support the Rx FIFO Full Configuration bit?
SIO4_FEATURE_BCR_SCD	Does the Board Control Register support the Single Cycle Disable bit?
SIO4_FEATURE_BSR	Is the Board Status Register supported?
SIO4_FEATURE_BSR_FIFO_PRESENT	Does the Board Status Register support the FIFO Present bits?
SIO4_FEATURE_COUNT	This reports the number of features supported by the service.
SIO4_FEATURE_FIFO_COUNT	Are the FIFO Count registers supported?
SIO4_FEATURE_FIFO_SIZE	Are the FIFO Size registers supported?
SIO4_FEATURE_FR	Is the Features Register supported?
SIO4_FEATURE_IRQ_32	Are all 32-bits of the interrupt configuration registers significant?
SIO4_FEATURE_MP	Is the Multi-Protocol transceiver feature in firmware?
SIO4_FEATURE_MP_CHIP	Which Multi-Protocol transceiver chip is present?
SIO4_FEATURE_MP_PROGRAM	Can a transceiver selection be reprogrammed?
SIO4_FEATURE_OSC_CHIP	Which programmable oscillator chip is present?
SIO4_FEATURE_OSC_MEASURE	Is the driver able to measure the oscillator's frequency?
SIO4_FEATURE_OSC_PER_CHAN	Is each channel separately and individually programmable?
SIO4_FEATURE_OSC_PROGRAM	Is the driver able to program the oscillator?
SIO4_FEATURE_PSRCR	Are the Pin Source Registers supported?
SIO4_FEATURE_PSTSR	Are the Pin Status Registers supported?
SIO4_FEATURE_SYNC_MODEL	Is this a SYNC model/version of the SIO4?
SIO4_FEATURE_Z16C30_MODEL	Is this a model that uses Zilog Z16C30 dual USC chips?

Macros (get only options)	Description
SIO4_FEATURE_NO	The feature is not supported.
SIO4_FEATURE_UNKNOWN	Either the feature is unknown or support for the feature is unknown.

`SIO4_FEATURE_YES`	The feature is supported.

Macros (other)	Description
`SIO4_FEATURE_LAST_INDEX`	This gives the highest defined feature index.

3.1.12. SIO4_INT_NOTIFY Bits

This set of macros defines the bits that may be combined to form the value passed to the `SIO4_INT_NOTIFY` IOCTL service.

> **WARNING:** If a USC interrupt occurs then that interrupt must be serviced within the USC by the application. If this is not done then that interrupt source within the USC will continue to function as an active USC interrupt source. In this case the SIO4 will continue to assert an interrupt while interrupts are appropriately enabled.

Macros	Description
`SIO4_INT_NOTIFY_RX_FIFO_AF`	The Rx FIFO Almost Full interrupt.
`SIO4_INT_NOTIFY_RX_FIFO_E`	The Rx FIFO Empty interrupt.
`SIO4_INT_NOTIFY_RX_FIFO_F`	The Rx FIFO Full interrupt.
`SIO4_INT_NOTIFY_SYNC_DETECTED`	The SYNC Detected interrupt.
`SIO4_INT_NOTIFY_TX_FIFO_AE`	The Tx FIFO Almost Empty interrupt.
`SIO4_INT_NOTIFY_TX_FIFO_E`	The Tx FIFO Empty interrupt.
`SIO4_INT_NOTIFY_TX_FIFO_F`	The Tx FIFO Full interrupt.
`SIO4_INT_NOTIFY_USC_INTERRUPTS`	The USC interrupts.

3.1.13. SIO4_IO_MODE_XXX Values

This set of macros defines the options available for the `SIO4_TX_IO_MODE_CONFIG` and `SIO4_RX_IO_MODE_CONFIG` IOCTL services.

Macros	Description
`SIO4_IO_MODE_DEFAULT`	This refers to the default I/O mode, which is PIO.
`SIO4_IO_MODE_DMA`	This refers to DMA, which is generally performed without regard to the FIFO's content.
`SIO4_IO_MODE_DMDMA`	This refers to Demand Mode DMA, which transfers data according to the FIFO's content.
`SIO4_IO_MODE_PIO`	This refers to PIO, which uses repetitive register accesses.
`SIO4_IO_MODE_READ`	When provided to the services, this requests the current setting.

3.1.14. SIO4_RX_CABLE_CONFIG Options

This set of macros defines the options available for the `SIO4_RX_CABLE_CONFIG` IOCTL service.

Macros (set and get options)	Description
`SIO4_RX_CABLE_CONFIG_DISABLE`	The receiver is disconnected from the cable.
`SIO4_RX_CABLE_CONFIG_LOWER`	The receiver is connected to the lower cable portion.
`SIO4_RX_CABLE_CONFIG_UPPER`	The receiver is connected to the upper cable portion.

Macros (set only options)	Description
`SIO4_RX_CABLE_CONFIG_READ`	Retrieve the current configuration.

Macros (get only options)	Description
`SIO4_RX_CABLE_CONFIG_INVALID`	The current configuration is invalid.

3.1.15. SIO4_RX_FIFO_FULL_CONFIG Options

This set of macros defines the options available for the `SIO4_RX_FIFO_FULL_CONFIG` IOCTL service.

Macros (set and get options)	Description
`SIO4_RX_FIFO_FULL_CONFIG_HALT`	Attempt to halt the flow of incoming data.
`SIO4_RX_FIFO_FULL_CONFIG_OVERRUN`	Let the FIFO overrun by discarding excess data.

Macros (set only options)	Description
`SIO4_RX_FIFO_FULL_CONFIG_READ`	Retrieve the current configuration.

3.1.16. SIO4_RXC_USC_CONFIG Options

This set of macros defines the options available for the `SIO4_RXC_USC_CONFIG` IOCTL service.

Macros (set and get options)	Description
`SIO4_RXC_USC_CONFIG_IN_CBL_RC`	An input from the cable's RxClk signal.
`SIO4_RXC_USC_CONFIG_IN_HI`	An input driven high.
`SIO4_RXC_USC_CONFIG_IN_LOW`	An input driven low.
`SIO4_RXC_USC_CONFIG_IN_PRG_CLK`	An input from the programmable clock.
`SIO4_RXC_USC_CONFIG_OUT_BRG0`	Output the BRG0 output signal.
`SIO4_RXC_USC_CONFIG_OUT_BRG1`	Output the BRG1 output signal.
`SIO4_RXC_USC_CONFIG_OUT_CTR1`	Output the CTR1 output signal.
`SIO4_RXC_USC_CONFIG_OUT_DPLL`	Output the DPLL output signal.
`SIO4_RXC_USC_CONFIG_OUT_TCC`	Output the USC's Transmit char clock signal.
`SIO4_RXC_USC_CONFIG_OUT_TCLK`	Output the USC's TxClk signal.
`SIO4_RXC_USC_CONFIG_OUT_TCOMP`	Output the USC Transmit Complete signal.

Macros (set only options)	Description
`SIO4_RXC_USC_CONFIG_READ`	Retrieve the current configuration.

Macros (get only options)	Description
`SIO4_RXC_USC_CONFIG_INVALID`	The current configuration is invalid.
`SIO4_RXC_USC_CONFIG_UNKNOWN`	The current configuration is unknown.

3.1.17. SIO4_TX_CABLE_CLOCK_CONFIG Options

This set of macros defines the options available for the `SIO4_TX_CABLE_CLOCK_CONFIG` IOCTL service.

Macros (set and get options)	Description
`SIO4_TX_CABLE_CLOCK_CONFIG_CBL_RC`	Output the cable's RxClk input signal.
`SIO4_TX_CABLE_CLOCK_CONFIG_DRV_HI`	An output driven high.
`SIO4_TX_CABLE_CLOCK_CONFIG_DRV_LOW`	An output driven low.
`SIO4_TX_CABLE_CLOCK_CONFIG_PRG_CLK`	Output the programmable clock output.
`SIO4_TX_CABLE_CLOCK_CONFIG_USC_RC`	Output the USC's RxClk output signal.
`SIO4_TX_CABLE_CLOCK_CONFIG_USC_TC`	Output the USC's TxClk output signal.

Macros (set only options)	Description
`SIO4_TX_CABLE_CLOCK_CONFIG_READ`	Retrieve the current configuration.

Macros (get only options)	Description
`SIO4_TX_CABLE_CLOCK_CONFIG_INVALID`	The current configuration is invalid.
`SIO4_TX_CABLE_CLOCK_CONFIG_UNKNOWN`	The current configuration is unknown.

3.1.18. SIO4_TX_CABLE_CONFIG Options

This set of macros defines the options available for the `SIO4_TX_CABLE_CONFIG` IOCTL service.

Macros (set and get options)	Description
`SIO4_TX_CABLE_CONFIG_BOTH`	The transmitter is connected to both the upper and the lower cable portions so that both are driven in parallel.
`SIO4_TX_CABLE_CONFIG_DISABLE`	Disconnect the transmitter from the cable.
`SIO4_TX_CABLE_CONFIG_LOWER`	The transmitter is connected to the lower cable portion.
`SIO4_TX_CABLE_CONFIG_UPPER`	The transmitter is connected to the upper cable portion.

Macros (set only options)	Description
`SIO4_TX_CABLE_CONFIG_READ`	Retrieve the current configuration.

Macros (get only options)	Description
`SIO4_TX_CABLE_CONFIG_INVALID`	The current configuration is invalid.

3.1.19. SIO4_TX_CABLE_DATA_CONFIG Options

This set of macros defines the options available for the `SIO4_TX_CABLE_DATA_CONFIG` IOCTL service (see page 100).

Macros (set and get options)	Description
`SIO4_TX_CABLE_DATA_CONFIG_DRV_LOW`	An output driven low.
`SIO4_TX_CABLE_DATA_CONFIG_DRV_HI`	An output driven high.
`SIO4_TX_CABLE_DATA_CONFIG_USC_TXD`	Output the USC's TxD output signal.

Macros (set only options)	Description
`SIO4_TX_CABLE_DATA_CONFIG_READ`	Retrieve the current configuration.

Macros (get only options)	Description
`SIO4_TX_CABLE_DATA_CONFIG_INVALID`	The current configuration is invalid.
`SIO4_TX_CABLE_DATA_CONFIG_UNKNOWN`	The current configuration is unknown.

3.1.20. SIO4_TXC_USC_CONFIG Options

This set of macros defines the options available for the `SIO4_TXC_USC_CONFIG` IOCTL service.

Macros (set and get options)	Description
`SIO4_TXC_USC_CONFIG_IN_CBL_RC`	An input from the cable's RxClk signal.
`SIO4_TXC_USC_CONFIG_IN_HI`	An input driven high.
`SIO4_TXC_USC_CONFIG_IN_LOW`	An input driven low.
`SIO4_TXC_USC_CONFIG_IN_PRG_CLK`	An input from the programmable clock.
`SIO4_TXC_USC_CONFIG_OUT_BRG0`	Output the BRG0 output signal.
`SIO4_TXC_USC_CONFIG_OUT_BRG1`	Output the BRG1 output signal.
`SIO4_TXC_USC_CONFIG_OUT_CTR1`	Output the CTR1 output signal.
`SIO4_TXC_USC_CONFIG_OUT_DPLL`	Output the DPLL output signal.
`SIO4_TXC_USC_CONFIG_OUT_TCC`	Output the USC's Transmit char clock signal.
`SIO4_TXC_USC_CONFIG_OUT_TCLK`	Output the USC's TxClk signal.
`SIO4_TXC_USC_CONFIG_OUT_TCOMP`	Output the USC Transmit Complete signal.

Macros (set only options)	Description
`SIO4_TXC_USC_CONFIG_READ`	Retrieve the current configuration.

Macros (get only options)	**Description**
`SIO4_TXC_USC_CONFIG_INVALID`	The current configuration is invalid.
`SIO4_TXC_USC_CONFIG_UNKNOWN`	The current configuration is unknown.

3.2. Data Types

This driver interface includes the following data types which are defined in `sio4.h`.

3.2.1. ADDR_SEARCH_MODE

This enumeration defines the receiver's possible address search mode options for use with the HDLC protocol.

Definition

```
typedef enum AddrSearchMode
{
    …
} ADDR_SEARCH_MODE;
```

Values	**Description**
`DISABLED`	Disable address search mode.
`EXT_PLUS_CTRL`	Search for an external address and one control byte.
`ONE_BYTE_NO_CTRL`	Search only for an address byte.
`ONE_BYTE_PLUS_CTRL`	Search for an address byte and a control byte.

3.2.2. BRG_MODE

This enumeration defines the Baud Rate Generator's possible operating modes.

Definition

```
typedef enum BRGMode
{
    …
} BRG_MODE;
```

Values	**Description**
`BRG_CONTINUOUS`	Count down continuously, reloading the starting value each time the count reaches zero.
`BRG_SINGLE_CYCLE`	Count down to zero one time only.

3.2.3. CHAR_LENGTH

This enumeration defines the receiver's and transmitter's possible data value sizes.

Definition

```
typedef enum CharLength
{
    …
} CHAR_LENGTH;
```

Values	**Description**
`BITS1`	Data values consist of one bit each.

BITS2	Data values consist of two bits each.
BITS3	Data values consist of three bits each.
BITS4	Data values consist of four bits each.
BITS5	Data values consist of five bits each.
BITS6	Data values consist of six bits each.
BITS7	Data values consist of seven bits each.
BITS8	Data values consist of eight bits each.

3.2.4. CLOCK_RATE

This enumeration defines the receiver's and transmitter's possible clock rate divisor options for use with the Asynchronous protocol. This effectively defines the length of each data bit.

Definition

```
typedef enum ClockRate
{
    …
} CLOCK_RATE;
```

Values	Description
LOCK_RATE	Reserved. Do not use.
RATE_X16	Divide the source clock by 16.
RATE_X32	Divide the source clock by 32.
RATE_X64	Divide the source clock by 64.

3.2.5. CLOCK_SOURCE

This enumeration defines all possible clock source options within each USC channel's clocking logic circuit. While the enumerated values are all-encompassing some clock source options are applicable only to certain clock source recipients.

Definition

```
typedef enum ClockSource
{
    …
} CLOCK_SOURCE;
```

Values	Description
BRG0_CLOCK	Select the BRG0 output as the clock source.
BRG1_CLOCK	Select the BRG1 output as the clock source.
CLOCK_DISABLED	Disable the clock recipient.
CTR0_CLOCK	Select the CTR0 output as the clock source.
CTR1_CLOCK	Select the CTR1 output as the clock source.
DPLL_CLOCK	Select the DPLL output as the clock source.
RXC_PIN_CLOCK	Select the RxC pin as the clock source.
TXC_PIN_CLOCK	Select the TxC pin as the clock source.

3.2.6. DATA_FORMAT

This enumeration defines the receiver's and transmitter's possible data encoding formats.

Definition

```
typedef enum DataFormat
{
    …
} DATA_FORMAT;
```

Values	Description
`BIPHASE_LEVEL`	Bi-Phase Level
`BIPHASE_MARK`	Bi-Phase Mark
`BIPHASE_SPACE`	Bi-Phase Space
`DIFF_BIPHASE_LEVEL`	Differential Bi-Phase Level
`NRZ`	Non-Return-to-Zero
`NRZB`	Inverted Non-Return-to-Zero
`NRZI_MARK`	Non-Return-to-Zero Invert Mark
`NRZI_SPACE`	Non-Return-to-Zero Invert Space

3.2.7. DPLL_DIVISOR

This enumeration defines the possible Digital Phase Lock Loop clock divisor options.

Definition

```
typedef enum DPLLDivisor
{
    …
} DPLL_DIVISOR;
```

Values	Description
`DPLL_16X`	Divide the source clock by 16.
`DPLL_32X`	Divide the source clock by 32.
`DPLL_8X`	Divide the source clock by eight.

3.2.8. DPLL_MODE

This enumeration defines the possible source data encoding category options for the Digital Phase Lock Loop.

Definition

```
typedef enum DPLLMode
{
    …
} DPLL_MODE;
```

Values	Description
`DPLL_BIPHASE_LEVEL`	A Differential Bi-Phase Level format.
`DPLL_BIPHASE_MARK_SPACE`	The Bi-Phase Mark Bi-Phase Space format.
`DPLL_DISABLED`	Disable the DPLL.
`DPLL_NRZ_NRZI`	A Non-Return-to-Zero format.

3.2.9. DPLL_RESYNC

This enumeration defines the possible Digital Phase Lock Loop resynchronization options.

Definition

```
typedef enum DPLLResync
{
    …
} DPLL_RESYNC;
```

Values	Description
`BOTH_EDGES`	Resynchronize on rising and falling edges.
`FALLING_EDGE`	Resynchronize on falling edges.
`RISING_EDGE`	Resynchronize on rising edges.
`SYNC_INHIBIT`	Run the DPLL continuously without synchronizing.

3.2.10. ENABLE_TYPE

This enumeration defines the possible receiver and transmitter enable/disable options.

Definition

```
typedef enum EnableType
{
    …
} ENABLE_TYPE;
```

Values	Description
`DISABLE_AFTER_TX_RX`	Disable as the end of the current message, frame or character.
`DISABLE_IMMED`	Disable immediately and unconditionally.
`ENABLE_WITH_AUTO`	Enable per DCD and CTS flow control pins.
`ENABLE_WO_AUTO`	Enable immediately.

3.2.11. FIFO_STATUS

This enumeration defines various possible values that may be received when reading a FIFO’s status.

NOTE: Other values are possible but are not seen in normal use.

NOTE: The Almost Empty status becomes active when the FIFO contains *ALMOST EMPTY* or fewer bytes. Here, *ALMOST EMPTY* refers to the value programmed into the lower 16 bits of the Tx and Rx Almost Registers.

NOTE: The Almost Full status becomes active when the FIFO can receive *ALMOST FULL* or fewer additional bytes before being full. Here, *ALMOST FULL* refers to the value programmed into the upper 16 bits of the Tx and Rx Almost Registers.

Definition

```
typedef enum FIFOStatus
{
    …
} FIFO_STATUS;
```

Values	Description
`ALMOST_EMPTY_STATUS`	The FIFO is almost full.
`ALMOST_FULL_STATUS`	The FIFO is almost full.
`EMPTY_STATUS`	The FIFO is empty.

`FULL_STATUS`	The FIFO is full.
`INVALID_STATUS`	The FIFO's current status is invalid.
`NOT_ALMOST_EMPTY_NOR_ALMOST_FULL_STATUS`	The FIFO level is between the almost full and the almost empty states.

3.2.12. IDLE_LINE_COND

This enumeration defines the possible transmitter pattern output options for what will be sent when the transmitter has no data to send.

Definition

```
typedef enum IdleLineCond
{
    …
} IDLE_LINE_COND;
```

Values	**Description**
`ALL_ONES_IDLE`	Send out all ones.
`ALL_ZEROS_IDLE`	Send out all zeroes.
`ALTERNATE_1_AND_0_IDLE`	Send out alternating ones and zeroes.
`ALTERNATE_MARK_AND_SPACE_IDLE`	Send out alternating marks and spaces.
`MARK_IDLE`	Send out all marks.
`RESERVED_IDLE`	Reserved. Do not use.
`SPACE_IDLE`	Send out all spaces.
`SYNC_FLAG_NORMAL_IDLE`	Send out the default for the selected protocol.

3.2.13. PARITY_TYPE

This enumeration defines the possible receiver and transmitter data parity options.

NOTE: Another component is used to enable or disable the use or parity.

Definition

```
typedef enum ParityType
{
    …
} PARITY_TYPE;
```

Values	**Description**
`EVEN_PARITY`	Utilize Even parity.
`ODD_PARITY`	Utilize Odd parity.
`MARK_PARITY`	Utilize Mark parity.
`SPACE_PARITY`	Utilize Space parity.

3.2.14. RCV_ASYNC_PROTOCOL

This structure defines the available receiver parameters for the Asynchronous protocol.

Definition

```
typedef struct RcvASYNCProtocol
{
    CLOCK_RATE  eRxClockRate;
} RCV_ASYNC_PROTOCOL;
```

Fields	Description
`eRxClockRate`	This configures the receiver's clock source divisor.

3.2.15. RCV_HDLC_PROTOCOL

This structure defines the available receiver parameters for the HDLC protocol.

Definition

```
typedef struct RcvHDLCProtocol
{
    ADDR_SEARCH_MODE    eAddrSearchMode;
    UINT8               u816BitControlEnable;
    UINT8               u8LogicalControlEnable;
} RCV_HDLC_PROTOCOL;
```

Fields	Description
`eAddrSearchMode`	This specifies the address search mode.
`u816BitControlEnable`	Use 16-bit control words for extended search.
`u8LogicalControlEnable`	Use logical controls for extended search.

3.2.16. REGISTER_MOD_PARAMS

This structure defines the data fields applicable to performing register read-modify-write operations with the `SIO4_MOD_REGISTER` IOCTL service.

Definition

```
typedef struct RegisterModParams
{
    UINT32  u32RegisterNumber;
    UINT32  u32Value;
    UINT32  u32Mask;
} REGISTER_MOD_PARAMS;
```

Fields	Description
`u32RegisterNumber`	This identifies the register to access.
`u32Value`	This is the value for the bits to modify.
`u32Mask`	This is the set of bits to modify. If a bit is set, then the bit from the above field is applied. If a bit is clear, then that register bit is unchanged.

3.2.17. REGISTER_PARAMS

This structure defines the data fields applicable to reading from and writing to SIO4 registers.

Definition

```
typedef struct RegisterParams
{
    UINT32  u32RegisterNumber;
    UINT32  u32Value;
} REGISTER_PARAMS;
```

Fields	Description
`u32RegisterNumber`	This identifies the register to access.
`u32Value`	This is either set to the value to write to the register or it records the value read.

3.2.18. SINT32

This data type is defined to be a signed 32-bit integer.

3.2.19. SIO4_CHAN_CMD

This enumeration defines the possible commands which may be submitted via the `SIO4_SEND_CHANNEL_COMMAND` IOCTL service.

Definition

```
typedef enum ChannelCmd
{
    …
} SIO4_CHAN_CMD;
```

Values	Description
`LOAD_RX_CHAR_CNT_CMD`	Load the Receive Character Count from the Receive Count Limit Register.
`LOAD_RX_TX_CHAR_CNT_CMD`	Perform both of the two above actions.
`LOAD_TX_CHAR_CNT_CMD`	Load the Transmit Character Count from the Transmit Count Limit Register.
`LOAD_TC0_CMD`	Load the Baud Rate Generator 0 counter from the Time Constant 0 Register
`LOAD_TC0_TC1_CMD`	Perform both of the two above actions.
`LOAD_TC1_CMD`	Load the Baud Rate Generator 1 counter from the Time Constant 1 Register
`NULL_CMD`	Perform no action at all. This equals a value of zero (0).
`RESET_HIGHEST_IUS`	Reset the highest Interrupt Under Service bit.
`RX_FIFO_PURGE_CMD`	Purge the USC channel's internal receive data FIFO.
`RX_PURGE_CMD`	Purge the USC channel's internal receive data and RCC FIFOs.
`RX_TX_FIFO_PURGE_CMD`	Perform both of the two above actions.
`SEL_LSB_FIRST_CMD`	Transmit and receive the Least Significant Bit first.
`SEL_MSB_FIRST_CMD`	Transmit and receive the Most Significant Bit first.
`SEL_STRAIGHT_CMD`	Do not use on the SIO4.
`SEL_SWAPPED_CMD`	Do not use on the SIO4.
`TRIG_CHAN_LOAD_DMA_CMD`	Do not use on the SIO4.
`TRIG_RX_DMA_CMD`	Initiate a USC to FIFO DMA transfer.
`TRIG_RX_TX_DMA_CMD`	Perform both of the two above actions.
`TRIG_TX_DMA_CMD`	Initiate a FIFO to USC DMA transfer.
`TX_FIFO_PURGE_CMD`	Purge the USC channel's internal transmit data FIFO.

3.2.20. SIO4_DRIVER_INFO

This structure defines the data fields for the information returned by the `SIO4_GET_DRIVER_INFO` IOCTL service.

Definition

```
typedef struct SIO4DriverInfo
{
    UINT8   u8VersionNumber[8];
    UINT8   u8Built[32];
} SIO4_DRIVER_INFO;
```

Fields	Description
`u8VersionNumber`	This field gives the driver version number as a string in the form of `X.XX`.
`u8Built`	This field gives the driver build date and time as a string. It is given in the C form of `printf("%s, %s", __DATE__, __TIME__)`.

3.2.21. SIO4_INIT_CHAN

This structure defines the data fields applicable to USC protocol independent initialization of a serial channel.

Definition

```
typedef struct SIO4InitChan
{
    SIO4_MODE        eMode;
    UINT32           u32BaudRate;
    ENABLE_TYPE      eRxEnable;
    DATA_FORMAT      eRxDataFormat;
    CHAR_LENGTH      eRxDataLength;
    UINT8            u8RxParityEnable;
    PARITY_TYPE      eRxParityType;
    ENABLE_TYPE      eTxEnable;
    DATA_FORMAT      eTxDataFormat;
    CHAR_LENGTH      eTxDataLength;
    UINT8            u8TxParityEnable;
    PARITY_TYPE      eTxParityType;
    IDLE_LINE_COND   eTxIdleLineCond;
    UINT8            u8TxWaitOnUnderrun;
    UINT8            u8EnableRxUpper;
    UINT8            u8EnableRxLower;
    UINT8            u8EnableTxUpper;
    UINT8            u8EnableTxLower;
    UINT16           u16TxAlmostEmpty;
    UINT16           u16TxAlmostFull;
    UINT16           u16RxAlmostEmpty;
    UINT16           u16RxAlmostFull;
    UINT8            u8EnableTxCableUpper;
    UINT8            u8EnableTxCableLower;
    UINT8            u8EnableRxCableUpper;
    UINT8            u8EnableRxCableLower;
} SIO4_INIT_CHAN;
```

Fields	Description
`eMode`	This specifies the communications protocol.
`u32BaudRate`	This specifies the baud rate.
`eRxEnable`	This specifies the receiver enable state.
`eRxDataFormat`	This specifies the receive data encoding format.
`eRxDataLength`	This specifies the receive data length in bits.

`u8RxParityEnable`	This enables or disable receiver parity. Zero (0) disables parity and one (1) enables it.
`eRxParityType`	This specifies the receiver parity type to use when receiver parity is enabled.
`eTxEnable`	This specifies the transmitter enable state.
`eTxDataFormat`	This specifies the transmit data encoding format.
`eTxDataLength`	This specifies the transmit data encoding format.
`u8TxParityEnable`	This enables or disable transmitter parity. Zero (0) disables parity and one (1) enables it.
`eTxParityType`	This specifies the transmitter parity type to use when receiver parity is enabled.
`eTxIdleLineCond`	This specifies the transmitter data patterm to be generated on an underun condition.
`u8TxWaitOnUnderrun`	A one (1) specifes that the transmitter is to wait for software to respond to an underrun condition. If zero (0) it does not wait.
`u8EnableRxUpper`	Unused.
`u8EnableRxLower`	Unused.
`u8EnableTxUpper`	Unused.
`u8EnableTxLower`	Unused.
`u16TxAlmostEmpty`	This specifies the level at which the external transmit FIFO reports the Almost Empty status.
`u16TxAlmostFull`	This specifies the level at which the external transmit FIFO reports the Almost Full status.
`u16RxAlmostEmpty`	This specifies the level at which the external receive FIFO reports the Almost Empty status.
`u16RxAlmostFull`	This specifies the level at which the external receive FIFO reports the Almost Full status.
`u8EnableTxCableUpper`	A value of one (1) enables the transmitter clock and data signals on the upper cable pins. A zero (0) disables them.
`u8EnableTxCableLower`	A value of one (1) enables the transmitter clock and data signals on the lower cable pins. A zero (0) disables them.
`u8EnableRxCableUpper`	A value of one (1) enables the receiver clock and data signals on the upper cable pins. A zero (0) disables them.
`u8EnableRxCableLower`	A value of one (1) enables the receiver clock and data signals on the lower cable pins. A zero (0) disables them.

3.2.22. SIO4_INTERRUPT_STATUS

This structure records the interrupt status bits from the SIO4 Interrupt Status Register for the current channel. The bits reflect the accumulated status since the last interrupt notification or status request.

Definition

```
typedef struct IntStatus
{
    UINT8   u8SIO4Status;
} SIO4_INTERRUPT_STATUS;
```

Fields	**Description**
`u8SIO4Status`	The channel’s interrupt status from the Interrupt Status Register. This may consist of either four or eight bits, depending on the board’s capabilities.

3.2.23. SIO4_MODE

This enumeration defines the possible USC data routing and test mode options.

Definition

```
typedef enum SIO4Mode
{
    …
} SIO4_MODE;
```

Fields	Description
`AUTO_ECHO`	Echo all receive data out the transmitter.
`EXT_LOCAL_LOOPBACK`	Route data through the USC's Local Loopback circuitry.
`INT_LOCAL_LOOPBACK`	Route data through the USC's Internal Local Loopback circuitry.
`NORMAL`	Route transmit data out the transmitter and receive data into the received.

3.2.24. sio4_mp_chip_t

This enumeration identifies the supported options for identifying the Multi-Protocol transceiver feature on the SIO4. The values are used in the `chip` field of the `sio4_mp_t` data structure, which is used with the Multi-Protocol transceiver based IOCTL services. Refers to the specific service for information on how this structure is used.

Definition

```
typedef enum
{
    …
} sio4_mp_chip_t;
```

Values	Description
`SIO4_MP_CHIP_FIXED`	This refers to a fixed protocol implementation. The driver may not know which protocol is implemented on the SIO4.
`SIO4_MP_CHIP_SP508`	This refers to the Sipex SP508 Multi-Protocol transceiver chip.
`SIO4_MP_CHIP_UNKNOWN`	The chip type is unknown.

3.2.25. sio4_mp_prot_t

This enumeration identifies the protocol options supported by the Multi-Protocol transceiver driver. The values are used in the `want` and `got` fields of the `sio4_mp_t` data structure, which is used with the Multi-Protocol transceiver based IOCTL services. Refers to the specific service for information on how this structure is used. Refer to the hardware user manual for detailed explanations of each protocol options.

Definition

```
typedef enum
{
    …
} sio4_mp_prot_t;
```

Values	Description
`SIO4_MP_PROT_DISABLE`	This refers to the disabled or tri-stated condition.
`SIO4_MP_PROT_INVALID`	This is returned by the driver when a requested protocol is unsupported or unrecognized.
`SIO4_MP_PROT_READ`	This requests that the driver report the current protocol.
`SIO4_MP_PROT_RS_232`	This refers to the RS-232 protocol.
`SIO4_MP_PROT_RS_422_485`	This refers to the RS-422/RS-485 protocols.
`SIO4_MP_PROT_RS_423`	This refers to the RS-423 protocol.

`SIO4_MP_PROT_UNKNOWN`	This is returned by the driver when the protocol is unknown.

3.2.26. sio4_mp_t

This data structure is used to exchange information and requests about the board's Multi-Protocol transceiver feature between applications and the driver. This structure is used with the Multi-Protocol transceiver based IOCTL services. Refers to the specific service for information on how this structure is used.

Definition

```
typedef struct
{
    __s32   chip;
    __s32   prot_want;
    __s32   prot_got;
} sio4_mp_t;
```

Field	**Description**
`chip`	The driver will fill this field in with the Multi-Protocol transceiver chip identifier. Refer to the `sio4_mp_chip_t` data type documentation elsewhere in this document.
`prot_want`	This refers to the protocol desired by the application.
`prot_got`	This refers to the protocol reported by the device.

3.2.27. sio4_osc_chip_t

This enumeration identifies the supported options for identifying the programmable oscillator feature on the SIO4. The values are used in the `chip` field of the `sio4_osc_t` data structure, which is used with the programmable oscillator based IOCTL services. Refers to the specific service for information on how this structure is used.

Definition

```
typedef enum
{
    …
} sio4_osc_chip_t;
```

Values	**Description**
`SIO4_OSC_CHIP_CY22393`	This refers to the Cypress CY22393, which provides each SIO4 channel with its own programmable oscillator.
`SIO4_OSC_CHIP_FIXED`	This refers to a fixed frequency, non-programmable oscillator that is shared by all SIO4 channels.
`SIO4_OSC_CHIP_IDC2053B`	This refers to a single Cypress IDC2053B, which provides all SIO4 channel with the same programmable oscillator output.
`SIO4_OSC_CHIP_IDC2053B_4`	This refers to four Cypress IDC2053B programmable oscillators, which provides each SIO4 channel with its own output.
`SIO4_OSC_CHIP_UNKNOWN`	The oscillator is unknown.

3.2.28. sio4_osc_t

This data structure is used to exchange information and requests about the board's programmable oscillator between applications and the driver. This structure is used with the programmable oscillator based IOCTL services. Refers to the specific service for information on how this structure is used.

Definition

```
typedef struct
{
    __u32   chip;
    __s32   freq_ref;
    __s32   freq_want;
    __s32   freq_got;
} sio4_osc_t;
```

Field	Description
`chip`	The driver will fill this field in with the oscillator chip identifier. Refer to the `sio4_osc_chip_t` data type documentation elsewhere in this document.
`freq_ref`	This refers to the frequency of the oscillator's reference source.
`freq_want`	This refers to the clock output frequency desired by the application.
`freq_got`	This refers to the clock output frequency produced by the device.

3.2.29. STATUS_BLOCK_OPTIONS

This enumeration defines the possible Receive Status Block and Transmit Control Block selection options for USC/FIFO DMA operations.

Definition

```
typedef enum StatusBlockOptions
{
    …
} STATUS_BLOCK_OPTIONS;
```

Fields	Description
`NO_STATUS_BLOCK`	Do not use Receive Status Blocks.
`ONE_WORD_STATUS_BLOCK`	Use 16-bit Receive Status Blocks.
`TWO_WORD_STATUS_BLOCK`	Use 32-bit Receive Status Blocks.

3.2.30. STOP_BITS

This enumeration defines the possible transmitter stop bit selection options for the Asynchronous protocol.

Definition

```
typedef enum StopBits
{
    …
} STOP_BITS;
```

Fields	Description
`ONE_STOP_BIT`	Use one stop bit.
`ONE_STOP_BIT_SHAVED`	Use one shaved stop bit.
`TWO_STOP_BITS`	Use two stop bits.
`TWO_STOP_BITS_SHAVED`	Use one full and one shaved stop bit.

NOTE: The number of 1/16 bits shaved is determined by the TxShaveL field of the USC Channel Command Register.

3.2.31. TX_RX

This enumeration defines the possible external FIFO reset selection options.

Definition

```
typedef enum TxRx
{
    …
} TX_RX;
```

Fields	Description
RX_FIFO	Reset the receive FIFO.
TX_FIFO	Reset the transmit FIFO.
TX_AND_RX_FIFO	Reset both FIFOs.

3.2.32. TX_UNDERRUN

This enumeration defines the overall set of possible transmitter under run responses for all supported protocols. The available options and their meanings vary with the protocol.

Definition

```
typedef enum TxUnderrun
{
    …
} TX_UNDERRUN;
```

Fields	Description
ABORT_COND	Send an abort.
CRC_FLAG_COND	Send a CRC then a flag.
EXT_ABORT_COND	Send a 16-bit abort.
FLAG_COND	Send a flag.

3.2.33. UINT16

This data type is defined to be an unsigned 16-bit integer.

3.2.34. UINT32

This data type is defined to be an unsigned 32-bit integer.

3.2.35. UINT8

This data type is defined to be an unsigned 8-bit integer.

3.2.36. USC_DMA_OPTIONS

This data structure defines the configurable parameters for DMA data transfer between the USC and the external FIFOs. The receiver and transmitter sides are independently configurable.

Definition

```
typedef struct
{
    STATUS_BLOCK_OPTIONS    eTxStatusBlockOptions;
    UINT8                   u8TxDMAWaitForTrigger;
    STATUS_BLOCK_OPTIONS    eRxStatusBlockOptions;
    UINT8                   u8RxDMAWaitForTrigger;
} USC_DMA_OPTIONS;
```

Field	Description
`eTxStatusBlockOptions`	Configure the use of transmitter status blocks.
`u8TxDMAWaitForTrigger`	Specifies when data transfer occurs.
`eRxStatusBlockOptions`	Configure the use of receiver status blocks.
`u8RxDMAWaitForTrigger`	Specifies when data transfer occurs.

3.2.37. XMT_ASYNC_PROTOCOL

This structure defines the available transmitter parameters for the Asynchronous protocol.

Definition

```
typedef struct XmtASYNCProtocol
{
    CLOCK_RATE  eTxClockRate;
    STOP_BITS   eTxStopBits;
} XMT_ASYNC_PROTOCOL;
```

Fields	Description
`eTxClockRate`	This specifies the source clock divisor.
`eTxStopBits`	This specifies the number of stop bits.

3.2.38. XMT_HDLC_PROTOCOL

This structure defines the available transmitter parameters for the HDLC protocol.

Definition

```
typedef struct XmtHDLCProtocol
{
    UINT8        u8SharedZeroFlags;
    UINT8        u8TxPreambleEnable;
    TX_UNDERRUN  eTxUnderrun;
} XMT_HDLC_PROTOCOL;
```

Fields	Description
`u8SharedZeroFlags`	This specifies that consecutive Flags do shared (1) or do not share (0) the zero.
`u8TxPreambleEnable`	This enables (1) or disables (0) sending of the preamble pattern.
`eTxUnderrun`	This specifies the transmitter response to an under run condition.

3.2.39. XMT_HDLC_SDLC_LOOP_PROTOCOL

This structure defines the available transmitter parameters for the HDLC/SDLC Loop protocol.

Definition

```
typedef struct XmtHDLCSDLCLoopProtocol
{
    UINT8        u8SharedZeroFlags;
    UINT8        u8TxActiveOnPoll;
    TX_UNDERRUN  eTxUnderrun;
} XMT_HDLC_SDLC_LOOP_PROTOCOL;
```

Fields	Description
u8SharedZeroFlags	This specifies that consecutive Flags do shared (1) or do not share (0) the zero.
u8TxActiveOnPoll	This specifies the disable/repeat (0) and inserts/send (1) options.
eTxUnderrun	This specifies the transmitter response to an under run condition.

3.2.40. XMT_ISOCHR_PROTOCOL

This structure defines the available transmitter parameters for the Isochronous protocol.

Definition

```
typedef struct XmtISOCHRProt
{
    UINT8   u8TwoStopBits;
}XMT_ISOCHR_PROTOCOL;
```

Fields	Description
u8TwoStopBits	A value of zero (0) specifies to use one stop bit and a value of one specified two stop bits.

3.3. Functions

This driver interface includes the following functions.

3.3.1. close()

This function is the entry point to close a connection to an open SIO4 serial channel. This function should only be called after a successful open of the respective device. Upon closing the channel, all settings and configurations are put in a reset state. The programmable oscillator reference frequency is unaffected.

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 "SIO4DocSrcLib.h"
```

```
int sio4_close(int fd, int verbose)
{
    int status;

    status  = close(fd);

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

    return(status);
}
```

3.3.2. ioctl()

This function is the entry point to performing setup and control operations on an open SIO4 serial channel. This function should only be called after a successful open of the respective device. The specific operation performed varies according to the `request` argument. The `request` argument also governs the use and interpretation of any additional arguments. The set of supported IOCTL services is defined in a following section.

Prototype

```
int ioctl(int fd, int request, ...);
```

Argument	Description
`fd`	This is the file descriptor of the device to access.
`request`	This specifies the desired operation to be performed.
`...`	This is any additional arguments. If `request` does not call for any additional arguments, then any additional arguments provided are ignored. The SIO4 IOCTL services use at most one argument, which is represented by a 32 bit value.

Return Value	Description
`-1`	An error occurred. Consult `errno`.
`0`	The operation succeeded.

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_ioctl(int fd, int request, int arg, int verbose)
{
    int status;

    status  = ioctl(fd, request, arg);

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

    return(status);
}
```

3.3.3. open()

This function is the entry point to open a connection to an SIO4 serial channel. Upon opening the channel, all settings and configurations are put in an initialized state. The programmable oscillator reference frequency is unaffected.

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 <assert.h>
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>

#include "SIO4DocSrcLib.h"

int sio4_open(int board, int channel, int verbose)
{
    int     fd;
    int     index;
    char    name[80];

    assert(board >= 0);
    assert((channel >= 0) && (channel <= 3));

    index   = (board * 4) + channel + 1;
    sprintf(name, "/dev/sio4%d", index);
    fd      = open(name, O_RDWR);

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

    return(fd);
}
```

3.3.4. read()

This function is the entry point to reading received data from an open SIO4 serial channel. This function should only be called after a successful open of the respective device. The function reads up to `count` bytes from the receive FIFO. If the number of bytes requested is not available within the configured time limit, the read operation times out.

> **NOTE:** Refer to the `SIO4_RX_IO_MODE_CONFIG` IOCTL services to configure this call for use of PIO, DMA or Demand Mode DMA data transfer.

Prototype

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

Argument	Description
`fd`	This is the file descriptor of the device to access.
`buf`	The data read will be put here.
`count`	This is the desired number of bytes to read.

Return Value	Description
`-1`	An error occurred. Consult `errno`.
`0` to `count`	The operation succeeded. If the return value is less than `count`, then the request timed out.

Example

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

#include "SIO4DocSrcLib.h"

int sio4_read(int fd, void *buf, size_t count, int verbose)
{
    int status;

    status  = read(fd, buf, count);

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

    return(status);
}
```

3.3.5. write()

This function is the entry point to writing data for transmission to an open SIO4 serial channel. This function should only be called after a successful open of the respective device. The function writes up to `count` bytes to the transmit FIFO. If the number of bytes requested cannot be sent within the configured time limit, the write operation times out.

> **NOTE:** Refer to the `SIO4_TX_IO_MODE_CONFIG` IOCTL services to configure this call for use of PIO, DMA or Demand Mode DMA data transfer.

Prototype

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

Argument	Description
`fd`	This is the file descriptor of the device to access.
`buf`	The data written comes from here.
`count`	This is the desired number of bytes to write.

Return Value	Description
`-1`	An error occurred. Consult `errno`.
`0` to `count`	The operation succeeded. If the return value is less than `count`, then the request timed out.

Example

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

#include "SIO4DocSrcLib.h"

int sio4_write(int fd, const void *buf, size_t count, int verbose)
{
    int status;

    status  = write(fd, buf, count);

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

    return(status);
}
```

3.4. IOCTL Services

The SIO4 driver implements the following IOCTL services. Each service is described along with the applicable `ioctl()` function arguments. In the definitions given the optional argument is identified as `arg` and is an unsigned 32-bit data type. Unless otherwise stated the return value definitions are those defined for the `ioctl()` function call.

NOTE: Many of the IOCTL services alter the state of the channel's operation and can adversely affect the channel's proper operation if data transfer is in progress. Exercise care when using these services to insure that data integrity is maintained.

3.4.1. SIO4_BOARD_JUMPERS

This service reads the jumper information for the user jumpers. If the jumpers are not supported on the board in use, then the returned value is the corresponding `XXX_UNKNOWN` macro. If the jumpers are supported, then the value returned will be from `0x0` to `0x3`.

Usage

ioctl() Argument	**Description**
request	SIO4_BOARD_JUMPERS
arg	__s32*

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_board_jumpers(int fd, __s32* get, int verbose)
{
    int status;

    get[0]  = 0xDEADBEEF;
    status  = ioctl(fd, SIO4_BOARD_JUMPERS, get);

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

    return(status);
}
```

3.4.2. SIO4_CABLE_CONFIG

This service configures the cable for the location where the cable data (TxD and RxD) and cable clock (TxClk and RxClk) signals will appear and retrieves the current configuration. If one of the predefined configurations is requested, it is applied. If the corresponding `XXX_READ` macro is supplied, then the current configuration is not changed. Before returning, the current configuration is obtained and reported to the caller. If the feature is unsupported, then the corresponding `XXX_UNKNOWN` macro is returned. If the feature is supported but the configuration is invalid, then the corresponding `XXX_INVALID` macro is returned.

Usage

ioctl() Argument	**Description**
request	SIO4_CABLE_CONFIG
arg	__s32* (see page 28 for valid values)

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_cable_config(int fd, __s32 set, __s32* get, int verbose)
{
    int status;
    status  = ioctl(fd, SIO4_CABLE_CONFIG, &set);

    if (get)
        get[0]  = set;

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

    return(status);
}
```

3.4.3. SIO4_CLEAR_DPLL_STATUS

This service clears status bits specific to the USC channel's Digital Phase Lock Loop. The specific values supported are given by macro definitions rather than an enumeration. These definitions are described earlier in this document.

Usage

`ioctl()` Argument	Description
`request`	`SIO4_CLEAR_DPLL_STATUS`
`arg`	`UINT32`

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_dpll_clear_status(int fd, int verbose)
{
    int status;

    status  = ioctl(fd,
                    SIO4_CLEAR_DPLL_STATUS,
                    CLEAR_DPLL_ALL_STATUS);

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

    return(status);
}
```

3.4.4. SIO4_CTS_CABLE_CONFIG

This service configures the cable's CTS signal and retrieves its current configuration. If one of the predefined configurations is requested, it is applied. If the corresponding `XXX_READ` macro is supplied, then the current configuration is not changed. Before returning, the current configuration is obtained and reported to the caller. If the feature is unsupported, then the corresponding `XXX_UNKNOWN` macro is returned. If the feature is supported but the configuration is invalid, then the corresponding `XXX_INVALID` macro is returned.

Usage

`ioctl()` **Argument**	**Description**
request	SIO4_CTS_CABLE_CONFIG
arg	`__s32*` (see page 29 for valid values)

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_cts_cable_config(int fd, __s32 set, __s32* get, int verbose)
{
    int status;

    status  = ioctl(fd, SIO4_CTS_CABLE_CONFIG, &set);

    if (get)
        get[0]  = set;

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

    return(status);
}
```

3.4.5. SIO4_DCD_CABLE_CONFIG

This service configures the cable's DCD signal and retrieves its current configuration. If one of the predefined configurations is requested, it is applied. If the corresponding `XXX_READ` macro is supplied, then the current configuration is not changed. Before returning, the current configuration is obtained and reported to the caller. If the feature is unsupported, then the corresponding `XXX_UNKNOWN` macro is returned. If the feature is supported but the configuration is invalid, then the corresponding `XXX_INVALID` macro is returned.

Usage

`ioctl()` **Argument**	**Description**
request	SIO4_DCD_CABLE_CONFIG
arg	`__s32*` (see page 29 for valid values)

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_dcd_cable_config(int fd, __s32 set, __s32* get, int verbose)
{
    int status;

    status  = ioctl(fd, SIO4_DCD_CABLE_CONFIG, &set);

    if (get)
        get[0]  = set;

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

    return(status);
}
```

3.4.6. SIO4_ENABLE_BRG0

This service enables or disables the USC channel's Baud Rate Generator 0. BRG0 is enabled by a value of `TRUE` and disabled by a value of `FALSE`.

Usage

`ioctl()` Argument	Description
`request`	`SIO4_ENABLE_BRG0`
`arg`	`UINT8`

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_brg0_enable(int fd, int enable, int verbose)
{
    int status;

    status  = ioctl(fd, SIO4_ENABLE_BRG0, enable);

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

    return(status);
}
```

3.4.7. SIO4_ENABLE_BRG1

This service enables or disables the USC channel's Baud Rate Generator 1. BRG1 is enabled by a value of `TRUE` and disabled by a value of `FALSE`.

Usage

`ioctl()` Argument	Description
`request`	`SIO4_ENABLE_BRG1`
`arg`	`UINT8`

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_brg1_enable(int fd, int enable, int verbose)
{
    int status;

    status  = ioctl(fd, SIO4_ENABLE_BRG1, enable);

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

    return(status);
}
```

3.4.8. SIO4_FEATURE_TEST

This service provides information on an SIO4's feature set. To gain support information on a specific feature the corresponding macro is supplied. The value returned will be the corresponding support information, which is the `XXX_YES` or `XXX_NO` macro in most cases. If the `XXX_COUNT` macro is supplied, the value returned is the number of feature options supported by the service, and should be one more that the service's `XXX_LAST_INDEX` macro.

Usage

`ioctl()` Argument	**Description**
request	SIO4_FEATURE_TEST
arg	__s32*

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_feature_test(int fd, __s32 set, __s32* get, int verbose)
{
    __s32   arg = set;
    int     status;

    status  = ioctl(fd, SIO4_FEATURE_TEST, &arg);

    if (get)
        get[0]  = arg;

    if ((verbose) && (status == -1))
        printf("ioctl() failure, errno = %d\n", errno);
    return(status);
}
```

3.4.9. SIO4_GET_DRIVER_INFO

This service retrieves information about the driver itself. At this time this includes only a driver version string.

Usage

`ioctl()` **Argument**	**Description**
`request`	`SIO4_GET_DRIVER_INFO`
`arg`	`SIO4_DRIVER_INFO*`

Example

```
    #include <errno.h>
    #include <stdio.h>
    #include <unistd.h>
    #include <sys/ioctl.h>

    #include "SIO4DocSrcLib.h"

    int sio4_driver_info_get(int fd, SIO4_DRIVER_INFO* info, int verbose)
    {
        int status;

        status  = ioctl(fd, SIO4_GET_DRIVER_INFO, info);

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

        return(status);
    }
```

3.4.10. SIO4_INIT_BOARD

This service initializes all of the board's hardware. This includes the USCs, the FIFOs, the cable configurations, the transceivers and the programmable oscillators. For boards with programmable oscillators and programmable transceivers, these features are initialized in preparation for use.

Usage

`ioctl()` **Argument**	**Description**
`request`	`SIO4_INIT_BOARD`
`arg`	Not used.

Example

```
    #include <errno.h>
    #include <stdio.h>
    #include <unistd.h>
    #include <sys/ioctl.h>

    #include "SIO4DocSrcLib.h"
    #include "SIO4DocSrcLib.h"
```

```
int sio4_board_init(int fd, int verbose)
{
    int status;

    status  = ioctl(fd, SIO4_INIT_BOARD);

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

    return(status);
}
```

3.4.11. SIO4_INIT_CHANNEL

This service initializes a channel by applying the settings given in the supplied structure. This service does not affect the transceivers or the programmable oscillator.

Usage

`ioctl()` Argument	**Description**
`request`	`SIO4_INIT_CHANNEL`
`arg`	`SIO4_INIT_CHAN*`

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_channel_init(int fd, int verbose)
{
    SIO4_INIT_CHAN  init;
    int             status;

    init.eMode                  = INT_LOCAL_LOOPBACK;
    init.u32BaudRate            = 1228800L;
    init.eRxEnable              = ENABLE_WO_AUTO;
    init.eRxDataFormat          = NRZ;
    init.eRxDataLength          = BITS8;
    init.u8RxParityEnable       = 0;
    init.eRxParityType          = EVEN_PARITY;
    init.eTxEnable              = ENABLE_WO_AUTO;
    init.eTxDataFormat          = NRZ;
    init.eTxDataLength          = BITS8;
    init.u8TxParityEnable       = 0;
    init.eTxParityType          = EVEN_PARITY;
    init.eTxIdleLineCond        = SYNC_FLAG_NORMAL_IDLE;
    init.u8TxWaitOnUnderrun     = 0;
    init.u8EnableRxUpper        = 1;
    init.u8EnableRxLower        = 0;
    init.u8EnableTxUpper        = 0;
    init.u8EnableTxLower        = 1;
    init.u16TxAlmostEmpty       = 1;
    init.u16TxAlmostFull        = 1;
    init.u16RxAlmostEmpty       = 1;
    init.u16RxAlmostFull        = 1;
    init.u8EnableTxCableUpper   = 0;
    init.u8EnableTxCableLower   = 1;
    init.u8EnableRxCableUpper   = 1;
    init.u8EnableRxCableLower   = 0;

    status  = ioctl(fd, SIO4_INIT_CHANNEL, &init);

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

    return(status);
}
```

3.4.12. SIO4_INT_NOTIFY

This service requests that the application be notified of one or more interrupts on the given serial channel. The parameter value is the bit wise or-ing of the possible notification bits. (The bits are defined in a previous section of this document.) Notification is given only for those bits which are set. Passing in a value of zero (`0`) cancels all notification requests. Once a specified interrupt occurs the driver clears and disables the interrupt, then notifies the application via a `SIGIO` (from `signal.h`) signal. To receive any subsequent notifications the application must make another notification request. The referenced interrupts are enabled. Unreferenced interrupts are disabled.

WARNING: If a USC interrupt occurs then that interrupt must be serviced within the USC by the application. If this is not done then that interrupt source within the USC will continue to function as an active USC interrupt source. In this case the SIO4 will continue to assert an interrupt while USC interrupts are enabled.

NOTE: The application will not receive notification of any interrupt that the driver itself is waiting on.

NOTE: This service automatically clears the current accumulated interrupt status.

NOTE: Interrupt options referenced but unsupported by the current hardware are quietly ignored.

Usage

`ioctl()` Argument	Description
`request`	`SIO4_INT_NOTIFY`
`arg`	`unsigned char`

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_int_notify(int fd, unsigned char notify, int verbose)
{
    int status;

    status  = ioctl(fd, SIO4_INT_NOTIFY, notify);

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

    return(status);
}
```

3.4.13. SIO4_MOD_REGISTER

This service performs a read-modify-write operation on an SIO4 register. This includes only the GSC firmware registers and USC registers. The PCI registers and the PLX feature set registers are read-only. Refer to the SIO4 User Manual and to `sio4.h` for a complete list of the available registers.

Usage

`ioctl()` Argument	**Description**
`request`	`SIO4_MOD_REGISTER`
`arg`	`REGISTER_MOD_PARAMS*`

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_reg_mod(
    int     fd,
    __u32   reg,
    __u32   value,
    __u32   mask,
    int     verbose)
{
    REGISTER_MOD_PARAMS parm;
    int             status;

    parm.u32RegisterNumber  = reg;
    parm.u32Value           = value;
    parm.u32Mask            = mask;
    status  = ioctl(fd, SIO4_MOD_REGISTER, &parm);

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

    return(status);
}
```

3.4.14. SIO4_NO_FUNCTION

This is an empty driver entry point. This IOCTL may be given to verify that the driver is correctly installed and that an SIO4 serial channel has been successfully opened. If an error status is returned then something isn't working properly.

Usage

`ioctl()` Argument	Description
`request`	`SIO4_NO_FUNCTION`
`arg`	Not used.

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_no_function(int fd, int verbose)
{
    int status;

    status  = ioctl(fd, SIO4_NO_FUNCTION);

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

    return(status);
}
```

3.4.15. SIO4_MP_CONFIG

This service is used to select and/or report on the current transceiver protocol. The driver uses the `prot_want` field and ignores all others. The results are recorded in the data structure's `prot_got` field. Refer to the Multi-Protocol transceiver programming information later in this document for more information.

> **NOTE:** The driver will fulfill the request based on the SIO4's capabilities. When the protocol can be changed and that requested is available, the requested change will be selected. Requests will otherwise fail and the protocol will be unchanged.

Usage

`ioctl()` Argument	Description
`request`	`SIO4_MP_CONFIG`
`arg`	`sio4_mp_t*`

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>
```

```
#include "SIO4DocSrcLib.h"

int sio4_mp_config(int fd, __s32 want, __s32* got, int verbose)
{
    sio4_mp_t   mp;
    int         status;

    mp.prot_want    = want;
    status          = ioctl(fd, SIO4_MP_CONFIG, &mp);

    if (got)
        got[0]  = mp.prot_got;

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

    return(status);
}
```

3.4.16. SIO4_MP_INFO

This service returns information about the current Multi-Protocol transceiver configuration. All field contents are ignored and are set by the driver according to the current configuration. Refer to the Multi-Protocol transceiver programming information later in this document for more information.

Usage

`ioctl()` Argument	Description
`request`	`SIO4_MP_INFO`
`arg`	`sio4_mp_t*`

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_mp_info(int fd, __u32* chip, __s32* got, int verbose)
{
    sio4_mp_t   mp;
    int         status;

    status  = ioctl(fd, SIO4_MP_INFO, &mp);

    if (chip)
        chip[0] = mp.chip;

    if (got)
        got[0]  = mp.prot_got;

    if ((verbose) && (status == -1))
        printf("ioctl() failure, errno = %d\n", errno);
    return(status);
}
```

3.4.17. SIO4_MP_INIT

This service initializes the board's Multi-Protocol transceiver feature. This returns the Multi-Protocol transceivers to their initial power up state. The results are recorded in the data structure's `prot_got` field. Refer to the Multi-Protocol transceiver programming information later in this document for more information.

Usage

`ioctl()` Argument	**Description**
`request`	`SIO4_MP_INIT`
`arg`	`sio4_mp_t*`

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_mp_init(int fd, __s32* got, int verbose)
{
    sio4_mp_t   mp;
    int         status;

    status  = ioctl(fd, SIO4_MP_INIT, &mp);

    if (got)
        got[0]  = mp.prot_got;

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

    return(status);
}
```

3.4.18. SIO4_MP_RESET

This service resets the board's Multi-Protocol transceiver feature. This disables the transceivers by tri-stating the outputs. The results are recorded in the data structure's `prot_got` field. Refer to the Multi-Protocol transceiver programming information later in this document for more information.

Usage

`ioctl()` Argument	**Description**
`request`	`SIO4_MP_RESET`
`arg`	`sio4_mp_t*`

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_mp_reset(int fd, __s32* got, int verbose)
{
    sio4_mp_t   mp;
    int         status;

    status  = ioctl(fd, SIO4_MP_RESET, &mp);

    if (got)
        got[0]  = mp.prot_got;

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

    return(status);
}
```

3.4.19. SIO4_MP_TEST

This service is used to determine if the board's Multi-Protocol transceiver feature supports a given protocol. The protocol to be tested is recorded in the structure's `prot_want` field. The results are recorded in the data structure's `prot_got` field. The reported value will be `SIO4_MP_PROT_INVALID` if the requested protocol value is unrecognized or unsupported. It will be `SIO4_MP_PROT_UNKNOWN` when support for the specified protocol is unknown. This is applicable when the SIO4 doesn't support the feature or when the chip used is unsupported by the driver. The reported value will equal the requested protocol when that protocol is supported. Refer to the Multi-Protocol transceiver programming information later in this document for more information.

Usage

`ioctl()` Argument	**Description**
`request`	`SIO4_MP_TEST`
`arg`	`sio4_mp_t*`

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_mp_test(int fd, __s32 want, __s32* got, int verbose)
{
    sio4_mp_t   mp;
    int         status;

    mp.prot_want    = want;
    status          = ioctl(fd, SIO4_MP_TEST, &mp);
    got[0]          = mp.prot_got;

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

    return(status);
}
```

3.4.20. SIO4_OSC_INFO

This service returns current configuration information about the channel's oscillator. The driver ignores the structure's current content and fills in all fields according to the channel's current configuration. Refer to the oscillator programming information later in this document for more information.

Usage

`ioctl()` Argument	**Description**
`request`	`SIO4_OSC_INFO`
`arg`	`sio4_osc_t*`

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_osc_info(
    int     fd,
    __u32*  chip,
    __s32*  freq_ref,
    __s32*  freq_want,
    __s32*  freq_got,
    int     verbose)
{
    sio4_osc_t  osc;
    int         status;

    status  = ioctl(fd, SIO4_OSC_INFO, &osc);

    if (chip)
        chip[0] = osc.chip;

    if (freq_ref)
        freq_ref[0] = osc.freq_ref;

    if (freq_want)
        freq_want[0]    = osc.freq_want;

    if (freq_got)
        freq_got[0] = osc.freq_got;

    if ((verbose) && (status == -1))
```

```
            printf("ioctl() failure, errno = %d\n", errno);

        return(status);
    }
```

3.4.21. SIO4_OSC_INIT

This service initializes the channel's programmable oscillator hardware. The channel's input clock will be reprogrammed to output the reference frequency as a result of this service, depending on the device's capabilities. The driver ignores the structure's current content and fills in all fields according to the channel's post-initialization configuration. The reference frequency is unaltered, the desired frequency is set to the reference frequency, and the frequency obtained is reported. Refer to the oscillator programming information later in this document for more information.

Usage

`ioctl()` Argument	Description
`request`	`SIO4_OSC_INIT`
`arg`	`sio4_osc_t*`

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_osc_init(int fd, __s32* freq_got, int verbose)
{
    sio4_osc_t  osc;
    int         status;

    status  = ioctl(fd, SIO4_OSC_INIT, &osc);

    if (freq_got)
        freq_got[0] = osc.freq_got;

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

    return(status);
}
```

3.4.22. SIO4_OSC_MEASURE

This service is used to measure the frequency produced by the current oscillator hardware configuration. The driver ignores all structure field values and fills them in according to the test results and the channel's current configuration. The test results are recorded in the data structure's `freq_got` field. A value of `-1` is reported when the frequency can't be measured. Refer to the oscillator programming information later in this document for more information.

> **NOTE:** The driver will perform a measurement test based on the SIO4's capabilities. When a measurement can be made, the test duration and the accuracy of the results are dependent on the board's capabilities. Refer to the hardware manual for additional details.

Usage

`ioctl()` Argument	Description
request	SIO4_OSC_MEASURE
arg	sio4_osc_t*

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_osc_measure(int fd, __s32* freq_got, int verbose)
{
    sio4_osc_t  osc;
    int         status;

    status      = ioctl(fd, SIO4_OSC_MEASURE, &osc);
    freq_got[0] = osc.freq_got;

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

    return(status);
}
```

3.4.23. SIO4_OSC_PROGRAM

This service is used to update and report on the programmed frequency produced by the channel's oscillator hardware. This service will reprogram the channel's oscillator hardware to produce the requested frequency, or one as near as possible to that requested. The resulting frequency will depend on the capability of the hardware and how its resources are being used, as applicable. If the requested value is `-1`, then the service will report the channel's current configuration without making any changes. The driver ignores all other fields and fills them in according to the channel's post-programming configuration. Refer to the oscillator programming information later in this document for more information.

Usage

`ioctl()` Argument	Description
request	SIO4_OSC_PROGRAM
arg	sio4_osc_t*

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_osc_program(int fd, __s32 want, __s32* got, int verbose)
```

```
{
    sio4_osc_t  osc;
    int         status;

    osc.freq_want   = want;
    status          = ioctl(fd, SIO4_OSC_PROGRAM, &osc);

    if (got)
        got[0]  = osc.freq_got;

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

    return(status);
}
```

3.4.24. SIO4_OSC_REFERENCE

This service is used to update and report on the recorded frequency for the channel's reference source. Changing this setting does not alter any existing programming results. New settings apply to subsequent calculations only! The only argument field used by the driver is the `freq_ref` field. If its value is `-1`, then the driver will report the current recorded reference frequency. The value supplied will otherwise be qualified per the requirements of the channel's oscillator and recorded for subsequent use. An error will be reported if it is invalid. The driver ignores all other fields and fills them in according to the channel's current configuration. This service does not alter any other oscillator related parameter. Refer to the oscillator programming information later in this document for more information.

> **CAUTION:** Setting the reference frequency to an incorrect value may have an adverse affect on the programmable oscillator. The results depend on the oscillator and the incorrect value specified.

Usage

`ioctl()` Argument	**Description**
`request`	`SIO4_OSC_REFERENCE`
`arg`	`sio4_osc_t*`

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_osc_reference(int fd, __s32* freq_ref, int verbose)
{
    sio4_osc_t  osc;
    int         status;

    osc.freq_ref    = freq_ref[0];
    status          = ioctl(fd, SIO4_OSC_REFERENCE, &osc);
    freq_ref[0]     = osc.freq_ref;

    if ((verbose) && (status == -1))
        printf("ioctl() failure, errno = %d\n", errno);
    return(status);
}
```

3.4.25. SIO4_OSC_RESET

This service resets the channel's oscillator hardware. The channel's input clock will be set to the lowest possible frequency as a result of this service, depending on the device's capabilities. The driver ignores the structure's current content and fills in all fields according to the channel's post-reset configuration. The reference frequency is unaltered, the desired frequency is set to zero, and the frequency obtained is reported. Refer to the oscillator programming information later in this document for more information.

Usage

`ioctl()` Argument	Description
`request`	`SIO4_OSC_RESET`
`arg`	`sio4_osc_t*`

Example

```
    #include <errno.h>
    #include <stdio.h>
    #include <unistd.h>
    #include <sys/ioctl.h>

    #include "SIO4DocSrcLib.h"

    int sio4_osc_reset(int fd, __s32* freq_got, int verbose)
    {
        sio4_osc_t  osc;
        int         status;

        status  = ioctl(fd, SIO4_OSC_RESET, &osc);

        if (freq_got)
            freq_got[0] = osc.freq_got;

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

        return(status);
    }
```

3.4.26. SIO4_OSC_TEST

This service reports the frequency that should be produced were the programming service requested for the desired frequency. The channel's input clock will be set to the lowest possible frequency as a result of this service, depending on the device's capabilities. The driver ignores the structure's current content and fills in all fields according to the channel's post-reset configuration. The reference frequency is unaltered, the desired frequency is set to zero, and the frequency obtained is reported. Refer to the oscillator programming information later in this document for more information.

Usage

ioctl() Argument	**Description**
request	SIO4_OSC_TEST
arg	sio4_osc_t*

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_osc_test(int fd, __s32 want, __s32* got, int verbose)
{
    sio4_osc_t  osc;
    int         status;

    osc.freq_want   = want;
    status          = ioctl(fd, SIO4_OSC_TEST, &osc);
    got[0]          = osc.freq_got;

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

    return(status);
}
```

3.4.27. SIO4_READ_FIFO_STATUS

This service reads the status of the channel's Tx and Rx FIFOs. The value reported by this service includes the status of both FIFOs packed into the lower eight bits of the parameter storage. This value must be decoded to obtain the proper status information.

Usage

ioctl() Argument	**Description**
request	SIO4_READ_FIFO_STATUS
arg	FIFO_STATUS*

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_fifo_read_status(
    int             fd,
    FIFO_STATUS*    tx,
    FIFO_STATUS*    rx,
    int             verbose)
```

```
{
    FIFO_STATUS fifo;
    int         status;

    status  = ioctl(fd, SIO4_READ_FIFO_STATUS, &fifo);

    if (tx)
        tx[0]   = 0xF & fifo;

    if (rx)
        rx[0]   = 0xF & (fifo >> 4);

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

    return(status);
}
```

3.4.28. SIO4_READ_INT_STATUS

This service requests the interrupt status information following interrupt notification. The status reported reflects all of the interrupts for the channel. The recorded status represents the accumulated status of all interrupts since the status was last read or notification requested. Once read, the recorded status is cleared.

> **WARNING:** If a USC interrupt occurs then that interrupt must be serviced within the USC by the application. If this is not done then that interrupt source within the USC will continue to function as an active USC interrupt source. In this case the SIO4 will continue to assert an interrupt while USC interrupts are enabled.
>
> **NOTE:** The application will not receive notification of any interrupt that the driver itself is waiting on.
>
> **NOTE:** Due to the timeliness of various interacting events it is possible for multiple interrupts to occur before the status is read. This can result in one SIGIO prompted status read reporting multiple interrupts and the next SIGIO prompted status read reporting no interrupts.

Usage

`ioctl()` Argument	Description
request	SIO4_READ_INT_STATUS
arg	SIO4_INTERRUPT_STATUS*

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_int_read_status(
    int                     fd,
    SIO4_INTERRUPT_STATUS*  int_stat,
    int                     verbose)
{
    int status;

    status  = ioctl(fd, SIO4_READ_INT_STATUS, int_stat);

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

    return(status);
}
```

3.4.29. SIO4_READ_REGISTER

This service reads the value of an SIO4 register. This includes all PCI registers, all PLX feature set registers, all GSC firmware registers, and all USC registers for the referenced channel. Refer to the SIO4 User Manual and to `sio4.h` for a complete list of the available registers.

Usage

`ioctl()` **Argument**	**Description**
`request`	`SIO4_READ_REGISTER`
`arg`	`REGISTER_PARAMS*`

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_reg_read(int fd, __u32 reg, __u32 *value, int verbose)
{
    REGISTER_PARAMS parm;
    int             status;

    parm.u32RegisterNumber  = reg;
    parm.u32Value           = 0xDEADBEEFL;
    status      = ioctl(fd, SIO4_READ_REGISTER, &parm);
    value[0]    = parm.u32Value;

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

    return(status);
}
```

3.4.30. SIO4_READ_REGISTER_RAW

This service reads the value of an SIO4 firmware register without respect to the channel being accessed. This applies to firmware registers only. Permissible values are from zero to 63. All other values result in failure. Refer to the SIO4 User Manual and to `sio4.h` for a complete list of the predefined register identifiers.

Usage

`ioctl()` Argument	Description
request	SIO4_READ_REGISTER_RAW
arg	REGISTER_PARAMS*

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_reg_read_raw(int fd, __u32 index, __u32 *value, int verbose)
{
    REGISTER_PARAMS parm;
    int             status;

    parm.u32RegisterNumber  = index;
    parm.u32Value           = 0xDEADBEEFL;
    status      = ioctl(fd, SIO4_READ_REGISTER_RAW, &parm);
    value[0]    = parm.u32Value;

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

    return(status);
}
```

3.4.31. SIO4_RESET_CHANNEL

This service performs a reset of the entire channel. This includes the USC, the FIFOs, the cable configuration, the transceivers and the programmable oscillator. The programmable transceivers and programmable oscillator are disabled, if supported in hardware. (The programmable oscillator is reset only if the SIO4 supports a different programmable source for each channel.)

Usage

`ioctl()` Argument	Description
request	SIO4_RESET_CHANNEL
arg	Not used.

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_channel_reset(int fd, int verbose)
{
    int status;

    status  = ioctl(fd, SIO4_RESET_CHANNEL);

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

    return(status);
}
```

3.4.32. SIO4_RESET_DEVICE

This service resets all of the board's hardware. This includes the USCs, the FIFOs, the cable configurations, the transceivers and the programmable oscillators. The programmable transceivers and programmable oscillators are disabled, if supported in hardware.

WARNING: This service affects all channels on the board and should be used with care.

Usage

`ioctl()` Argument	Description
`request`	`SIO4_RESET_DEVICE`
`arg`	Not used.

Example

```
    #include <errno.h>
    #include <stdio.h>
    #include <unistd.h>
    #include <sys/ioctl.h>

    #include "SIO4DocSrcLib.h"

    int sio4_reset_device(int fd, int verbose)
    {
        int status;

        status  = ioctl(fd, SIO4_RESET_DEVICE);

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

        return(status);
    }
```

3.4.33. SIO4_RESET_FIFO

This service resets either or both of the channel FIFOs.

Usage

`ioctl()` Argument	Description
`request`	`SIO4_RESET_FIFO`
`arg`	`TX_RX`

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_fifo_reset(int fd, TX_RX which, int verbose)
{
    int status;

    status  = ioctl(fd, SIO4_RESET_FIFO, which);

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

    return(status);
}
```

3.4.34. SIO4_RESET_USC

This service performs a reset of the channel's USC. The FIFOs, the cable configuration and the programmable oscillators are unaffected. This service has no affect on any other channels.

Usage

`ioctl()` Argument	Description
`request`	`SIO4_RESET_USC`
`arg`	Not used.

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_usc_reset(int fd, int verbose)
{
    int status;

    status  = ioctl(fd, SIO4_RESET_USC);

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

    return(status);
}
```

3.4.35. SIO4_RESET_ZILOG_CHIP

This service resets the entire Zilog Z16C30 dual USC. The reset is implemented using the chips hardware reset feature, which resets the referenced serial channel and the chip's other channel. If the other channel is in use the reset may interfere with its operation. The FIFOs, programmable oscillators and the cable configurations are unaffected.

> **WARNING:** This IOCTL resets both Z16C30 serial channels. Requesting this service may adversely affect the application or thread using the chip's alternate channel. A safe alternative is to use the `SIO4_RESET_USC` IOCTL service.

Usage

`ioctl()` Argument	**Description**
`request`	`SIO4_RESET_ZILOG_CHIP`
`arg`	Not used.

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_zilog_reset(int fd, int verbose)
{
    int status;

    status  = ioctl(fd, SIO4_RESET_ZILOG_CHIP);

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

    return(status);
}
```

3.4.36. SIO4_RX_CABLE_CONFIG

This service configures the receiver's connection to the cable interface and retrieves its current configuration. If one of the predefined configurations is requested, it is applied. If the corresponding `XXX_READ` macro is supplied, then the current configuration is not changed. Before returning, the current configuration is obtained and reported to the caller. If the configuration is invalid, then the corresponding `XXX_INVALID` macro is returned.

Usage

`ioctl()` Argument	**Description**
`request`	`SIO4_RX_CABLE_CONFIG`
`arg`	`__s32*`

Example

```
#include <errno.h>
#include <stdio.h>
```

```
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_rx_cable_config(int fd, __s32 set, __s32* get, int verbose)
{
    int status;

    status  = ioctl(fd, SIO4_RX_CABLE_CONFIG, &set);

    if (get)
        get[0]  = set;

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

    return(status);
}
```

3.4.37. SIO4_RX_FIFO_AE_CONFIG

This service configures the Rx FIFO Almost Empty level and reports the current level. When applying a setting, the Rx FIFO is reset and the current content is lost. If the corresponding `XXX_READ` macro is supplied then no change is applied. Before returning the current programmed level is obtained and supplied to the caller.

Usage

`ioctl()` Argument	**Description**
`request`	`SIO4_RX_FIFO_AE_CONFIG`
`arg`	`__s32*`

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_rx_fifo_ae_config(int fd, __s32 set, __s32* get, int verbose)
{
    int status;

    status  = ioctl(fd, SIO4_RX_FIFO_AE_CONFIG, &set);

    if (get)
        get[0]  = set;

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

    return(status);
}
```

3.4.38. SIO4_RX_FIFO_AF_CONFIG

This service configures the Rx FIFO Almost Full level and reports the current level. When applying a setting, the Rx FIFO is reset and the current content is lost. If the corresponding `XXX_READ` macro is supplied then no change is applied. Before returning the current programmed level is obtained and supplied to the caller.

Usage

`ioctl()` Argument	**Description**
`request`	`SIO4_RX_FIFO_AF_CONFIG`
`arg`	`__s32*`

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_rx_fifo_af_config(int fd, __s32 set, __s32* get, int verbose)
{
    int status;

    status  = ioctl(fd, SIO4_RX_FIFO_AF_CONFIG, &set);

    if (get)
        get[0]  = set;

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

    return(status);
}
```

3.4.39. SIO4_RX_FIFO_COUNT

This service retrieves the current Rx FIFO fill level. The value obtained is either the number of bytes of data in the Rx FIFO or the corresponding `XXX_UNKNOWN` macro if the Rx FIFO Count Register is unsupported.

Usage

`ioctl()` Argument	**Description**
`request`	`SIO4_RX_FIFO_COUNT`
`arg`	`__s32*`

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"
```

```
int sio4_rx_fifo_count(int fd, __s32* get, int verbose)
{
    int status;

    status  = ioctl(fd, SIO4_RX_FIFO_COUNT, get);

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

    return(status);
}
```

3.4.40. SIO4_RX_FIFO_FULL_CONFIG

This service configures how the receiver responds to an Rx FIFO Full condition and reports on the current configuration. If one of the predefined configurations is requested, it is applied. If the corresponding `XXX_READ` macro is supplied, then the current configuration is not changed. Before returning, the current configuration is obtained and reported to the caller. If the feature is not configurable on the current board, then no change can be applied.

Usage

`ioctl()` Argument	**Description**
`request`	`SIO4_RX_FIFO_FULL_CONFIG`
`arg`	`__s32*`

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_rx_fifo_full_config(
    int     fd,
    __s32   set,
    __s32*  get,
    int     verbose)
{
    int status;

    status  = ioctl(fd, SIO4_RX_FIFO_FULL_CONFIG, &set);

    if (get)
        get[0]  = set;

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

    return(status);
}
```

3.4.41. SIO4_RX_FIFO_SIZE

This service retrieves the size of the Rx FIFO. The value obtained is either the capacity of the Rx FIFO in bytes or the corresponding `XXX_UNKNOWN` macro if the Rx FIFO Size Register is unsupported.

Usage

`ioctl()` Argument	**Description**
`request`	`SIO4_RX_FIFO_SIZE`
`arg`	`__s32*`

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_rx_fifo_size(int fd, __s32* get, int verbose)
{
    int status;

    status  = ioctl(fd, SIO4_RX_FIFO_SIZE, get);

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

    return(status);
}
```

3.4.42. SIO4_RX_FIFO_TYPE

This service reports the Rx FIFO's firmware/hardware type implementation. The value obtained is either the FIFO's type or the corresponding `XXX_UNKNOWN` macro if the type cannot be determined.

Usage

`ioctl()` Argument	**Description**
`request`	`SIO4_RX_FIFO_TYPE`
`arg`	`__s32*`

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_rx_fifo_type(int fd, __s32* get, int verbose)
{
    int status;
    status  = ioctl(fd, SIO4_RX_FIFO_TYPE, get);

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

    return(status);
}
```

3.4.43. SIO4_RX_IO_ABORT

This service aborts an active data read operation (an active `read()` request). This has no affect if no such request is active and it has no affect on future requests.

Usage

`ioctl()` Argument	Description
`request`	`SIO4_RX_IO_ABORT`
`arg`	Unused

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_rx_io_abort(int fd, int verbose)
{
    int status;

    status  = ioctl(fd, SIO4_RX_IO_ABORT);

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

    return(status);
}
```

3.4.44. SIO4_RX_IO_MODE_CONFIG

This service updates and reports the mode used by the driver for data read operations. This refers to how data is moved from the SIO4 to host memory when the `read()` function is called.

Usage

`ioctl()` Argument	Description
`request`	`SIO4_RX_IO_MODE_CONFIG`
`arg`	`__s32*`

Example

```
#include <errno.h>
```

```
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_rx_io_mode_config(int fd, __s32 set, __s32* get, int verbose)
{
    int status;

    status  = ioctl(fd, SIO4_RX_IO_MODE_CONFIG, &set);

    if (get)
        get[0]  = set;

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

    return(status);
}
```

3.4.45. SIO4_RXC_USC_CONFIG

This service configures the channel's use of the USC RxC signal and retrieves its current configuration. If one of the predefined configurations is requested, it is applied. If the corresponding `XXX_READ` macro is supplied, then the current configuration is not changed. Before returning, the current configuration is obtained and reported to the caller. If the feature is unsupported, then the corresponding `XXX_UNKNOWN` macro is returned. If the feature is supported but the configuration is invalid, then the corresponding `XXX_INVALID` macro is returned.

Usage

`ioctl()` Argument	**Description**
`request`	`SIO4_RXC_USC_CONFIG`
`arg`	`__s32*`

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_rxc_usc_config(int fd, __s32 set, __s32* get, int verbose)
{
    int status;

    status  = ioctl(fd, SIO4_RXC_USC_CONFIG, &set);

    if (get)
        get[0]  = set;

    if ((verbose) && (status == -1))
        printf("ioctl() failure, errno = %d\n", errno);
    return(status);
}
```

3.4.46. SIO4_SELECT_DPLL_RESYNC

This service sets the resynchronization option for the USC channel's Digital Phase Lock Loop.

Usage

`ioctl()` Argument	Description
request	SIO4_SELECT_DPLL_RESYNC
arg	DPLL_RESYNC

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_dpll_set_resync(int fd, DPLL_RESYNC resync, int verbose)
{
    int status;

    status  = ioctl(fd, SIO4_SELECT_DPLL_RESYNC, resync);

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

    return(status);
}
```

3.4.47. SIO4_SEND_CHANNEL_COMMAND

This service sends a command to the channel's command register.

Usage

`ioctl()` Argument	Description
request	SIO4_SEND_CHANNEL_COMMAND
arg	SIO4_CHAN_CMD

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_channel_command_send(
    int             fd,
    SIO4_CHAN_CMD   command,
    int             verbose)
{
    int status;

    status  = ioctl(fd, SIO4_SEND_CHANNEL_COMMAND, command);

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

    return(status);
}
```

3.4.48. SIO4_SET_BRG0_MODE

This service sets the operating mode for the USC channel's Baud Rate Generator 0.

Usage

`ioctl()` Argument	Description
`request`	`SIO4_SET_BRG0_MODE`
`arg`	`BRG_MODE`

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_brg0_set_mode(int fd, BRG_MODE mode, int verbose)
{
    int status;

    status  = ioctl(fd, SIO4_SET_BRG0_MODE, mode);

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

    return(status);
}
```

3.4.49. SIO4_SET_BRG0_SOURCE

This service sets the USC channel's Baud Rate Generator 0 clock source. The only `CLOCK_SOURCE` enumeration values that are valid options for this IOCTL are those listed below.

- Counter 0
- Counter 1
- The RxC pin

- The TxC pin

Usage

`ioctl()` Argument	Description
request	SIO4_SET_BRG0_SOURCE
arg	CLOCK_SOURCE

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_brg0_set_clock_source(int fd, CLOCK_SOURCE src, int verbose)
{
    int status;

    status  = ioctl(fd, SIO4_SET_BRG0_SOURCE, src);

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

    return(status);
}
```

3.4.50. SIO4_SET_BRG1_MODE

This service sets the operating mode for the USC channel's Baud Rate Generator 1.

Usage

`ioctl()` Argument	Description
request	SIO4_SET_BRG1_MODE
arg	BRG_MODE

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_brg1_set_mode(int fd, BRG_MODE mode, int verbose)
{
    int status;

    status  = ioctl(fd, SIO4_SET_BRG1_MODE, mode);

    if ((verbose) && (status == -1))
```

```
            printf("ioctl() failure, errno = %d\n", errno);

        return(status);
    }
```

3.4.51. SIO4_SET_BRG1_SOURCE

This service sets the USC channel's Baud Rate Generator 1 clock source. The only `CLOCK_SOURCE` enumeration values that are valid options for this IOCTL are those listed below.

- Counter 0
- Counter 1
- The RxC pin
- The TxC pin

Usage

`ioctl()` Argument	**Description**
`request`	`SIO4_SET_BRG1_SOURCE`
`arg`	`CLOCK_SOURCE`

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_brg1_set_clock_source(int fd, CLOCK_SOURCE src, int verbose)
{
    int status;

    status  = ioctl(fd, SIO4_SET_BRG1_SOURCE, src);

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

    return(status);
}
```

3.4.52. SIO4_SET_CTR0_SOURCE

This service sets the USC channel's Counter 0 clock source. The only `CLOCK_SOURCE` enumeration values that are valid options for this IOCTL are those listed below.

- Disable the counter
- The RxC pin
- The TxC pin

Usage

`ioctl()` Argument	**Description**
`request`	`SIO4_SET_CTR0_SOURCE`
`arg`	`CLOCK_SOURCE`

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_ctr0_set_clock_source(int fd, CLOCK_SOURCE src, int verbose)
{
    int status;

    status  = ioctl(fd, SIO4_SET_CTR0_SOURCE, src);

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

    return(status);
}
```

3.4.53. SIO4_SET_CTR1_SOURCE

This service sets the USC channel's Counter 1 clock source. The only `CLOCK_SOURCE` enumeration values that are valid options for this IOCTL are those listed below.

- Disable the counter
- The RxC pin
- The TxC pin

Usage

`ioctl()` Argument	**Description**
`request`	`SIO4_SET_CTR1_SOURCE`
`arg`	`CLOCK_SOURCE`

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_ctr1_set_clock_source(int fd, CLOCK_SOURCE src, int verbose)
{
    int status;

    status  = ioctl(fd, SIO4_SET_CTR1_SOURCE, src);

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

    return(status);
}
```

3.4.54. SIO4_SET_USC_DMA_OPTIONS

This service configures the USC channel's DMA feature for data transfer between the USC and the external FIFOs. In addition to configuring the parameters referenced by the structure, this service configures the necessary USC I/O pins to permit proper USC/FIFO DMA data transfer.

Usage

`ioctl()` Argument	**Description**
`request`	`SIO4_SET_USC_DMA_OPTIONS`
`arg`	`DMA_OPTIONS*`

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_dma_set_options(
    int                     fd,
    STATUS_BLOCK_OPTIONS    tx_opts,
    UINT8                   tx_wait,
    STATUS_BLOCK_OPTIONS    rx_opts,
    UINT8                   rx_wait,
    int                     verbose)
{
    USC_DMA_OPTIONS parm;
    int         status;

    parm.eTxStatusBlockOptions  = tx_opts;
    parm.u8TxDMAWaitForTrigger  = tx_wait;
    parm.eRxStatusBlockOptions  = rx_opts;
    parm.u8RxDMAWaitForTrigger  = rx_wait;
    status  = ioctl(fd, SIO4_SET_USC_DMA_OPTIONS, &parm);

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

    return(status);
}
```

3.4.55. SIO4_SET_DPLL_DIVISOR

This service sets the clock source divisor used by the USC channel's Digital Phase Lock Loop.

Usage

`ioctl()` **Argument**	**Description**
request	SIO4_SET_DPLL_DIVISOR
arg	DPLL_DIVISOR

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_dpll_set_divisor(int fd, DPLL_DIVISOR divisor, int verbose)
{
    int status;

    status  = ioctl(fd, SIO4_SET_DPLL_DIVISOR, divisor);

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

    return(status);
}
```

3.4.56. SIO4_SET_DPLL_MODE

This service sets the encoding format used by the data input signal to the USC channel's Digital Phase Lock Loop.

Usage

`ioctl()` **Argument**	**Description**
request	SIO4_SET_DPLL_MODE
arg	DPLL_MODE

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_dpll_set_mode(int fd, DPLL_MODE mode, int verbose)
{
    int status;

    status  = ioctl(fd, SIO4_SET_DPLL_MODE, mode);
    if ((verbose) && (status == -1))
        printf("ioctl() failure, errno = %d\n", errno);

    return(status);
}
```

3.4.57. SIO4_SET_DPLL_SOURCE

This service sets the USC channel's Digital Phase Lock Loop clock source. The only `CLOCK_SOURCE` enumeration values that are valid options for this IOCTL are those listed below.

- Baud Rate Generator 0
- Baud Rate Generator 1
- The RxC pin
- The TxC pin

Usage

`ioctl()` Argument	**Description**
`request`	`SIO4_SET_DPLL_SOURCE`
`arg`	`CLOCK_SOURCE`

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_dpll_set_clock_source(int fd, CLOCK_SOURCE src, int verbose)
{
    int status;

    status  = ioctl(fd, SIO4_SET_DPLL_SOURCE, src);

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

    return(status);
}
```

3.4.58. SIO4_SET_RCV_ASYNC_PROT

This service configures the receiver specific Asynchronous parameters.

Usage

`ioctl()` Argument	**Description**
`request`	`SIO4_SET_RCV_ASYNC_PROT`

arg	RCV_ASYNC_PROTOCOL*

Example

```
/* This software is covered by the GNU GENERAL PUBLIC LICENSE (GPL). */
#include <errno.h>
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_async_setup_rx(int fd, CLOCK_RATE rate, int verbose)
{
    RCV_ASYNC_PROTOCOL  parm;
    int                 status;

    parm.eRxClockRate   = rate;
    status  = ioctl(fd, SIO4_SET_RCV_ASYNC_PROT, &parm);

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

    return(status);
}
```

3.4.59. SIO4_SET_RCV_HDLC_PROT

This service configures the receiver specific HDLC parameters.

Usage

`ioctl()` Argument	Description
request	SIO4_SET_RCV_HDLC_PROT
arg	RCV_HDLC_PROTOCOL*

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_hdlc_setup_rx(int fd, int verbose)
{
    RCV_HDLC_PROTOCOL   parm;
    int                 status;

    parm.eAddrSearchMode        = ONE_BYTE_NO_CTRL;
    parm.u816BitControlEnable   = 1;
    parm.u8LogicalControlEnable = 0;
    status  = ioctl(fd, SIO4_SET_RCV_HDLC_PROT, &parm);
    if ((verbose) && (status == -1))
        printf("ioctl() failure, errno = %d\n", errno);

    return(status);
}
```

3.4.60. SIO4_SET_RCV_ISOCHR_PROT

This service configures the receiver specific Isochronous parameters.

Usage

`ioctl()` Argument	Description
`request`	`SIO4_SET_RCV_ISOCHR_PROT`
`arg`	Not used.

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_isochr_setup_rx(int fd, int verbose)
{
    int status;

    status  = ioctl(fd, SIO4_SET_RCV_ISOCHR_PROT);

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

    return(status);
}
```

3.4.61. SIO4_SET_READ_TIMEOUT

This service sets the timeout limit for read requests, and is the maximum amount of time the driver will wait for a blocking `read()` request to complete. The timeout period is specified in seconds. Timeout values of zero (0) or less mean do not wait.

Usage

`ioctl()` Argument	Description
`request`	`SIO4_SET_READ_TIMEOUT`
`arg`	`SINT32`

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>
```

```
#include "SIO4DocSrcLib.h"

int sio4_timeout_set_read(int fd, UINT32 timeout, int verbose)
{
    int status;

    status  = ioctl(fd, SIO4_SET_READ_TIMEOUT, timeout);

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

    return(status);
}
```

3.4.62. SIO4_SET_RX_CLOCK_SOURCE

This service sets the receive clock source within the channel's USC. This applies to signal routing inside the USC only. All of the `CLOCK_SOURCE` enumeration values are valid options for this IOCTL.

Usage

`ioctl()` Argument	**Description**
request	SIO4_SET_RX_CLOCK_SOURCE
arg	CLOCK_SOURCE

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_rx_set_clock_source(int fd, CLOCK_SOURCE source, int verbose)
{
    int status;

    status  = ioctl(fd, SIO4_SET_RX_CLOCK_SOURCE, source);

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

    return(status);
}
```

3.4.63. SIO4_SET_SYNC_BYTE

This service sets the value of the `SYNC_CHARACTER` register, which can be used to trigger an interrupt when a byte of that value is moved into the channel's Rx FIFO.

Usage

`ioctl()` Argument	Description
request	SIO4_SET_SYNC_BYTE
arg	UINT8

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_sync_byte_set(int fd, UINT8 byte, int verbose)
{
    int status;

    status  = ioctl(fd, SIO4_SET_SYNC_BYTE, byte);

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

    return(status);
}
```

3.4.64. SIO4_SET_TX_CLOCK_SOURCE

This service sets the transmit clock source within the channel's USC. This applies to signal routing inside the USC only. All of the `CLOCK_SOURCE` enumeration values are valid options for this IOCTL.

Usage

`ioctl()` Argument	Description
request	SIO4_SET_TX_CLOCK_SOURCE
arg	CLOCK_SOURCE

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_tx_set_clock_source(int fd, CLOCK_SOURCE source, int verbose)
{
    int status;

    status  = ioctl(fd, SIO4_SET_TX_CLOCK_SOURCE, source);

    if ((verbose) && (status == -1))
        printf("ioctl() failure, errno = %d\n", errno);
    return(status);
}
```

3.4.65. SIO4_SET_WRITE_TIMEOUT

This service sets the timeout limit for write requests, and is the maximum amount of time the driver will wait for a blocking `write()` request to complete. The timeout period is specified in seconds. Timeout values of zero (0) or less mean do not wait.

Usage

`ioctl()` Argument	**Description**
`request`	`SIO4_SET_WRITE_TIMEOUT`
`arg`	`SINT32`

Example

```
    #include <errno.h>
    #include <stdio.h>
    #include <unistd.h>
    #include <sys/ioctl.h>

    #include "SIO4DocSrcLib.h"

    int sio4_timeout_set_write(int fd, UINT32 timeout, int verbose)
    {
        int status;

        status  = ioctl(fd, SIO4_SET_WRITE_TIMEOUT, timeout);

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

        return(status);
    }
```

3.4.66. SIO4_SET_XMT_ASYNC_PROT

This service configures the transmitter specific Asynchronous parameters.

Usage

`ioctl()` Argument	**Description**
`request`	`SIO4_SET_XMT_ASYNC_PROT`
`arg`	`XMT_ASYNC_PROTOCOL*`

Example

```
    #include <errno.h>
    #include <stdio.h>
    #include <unistd.h>
    #include <sys/ioctl.h>

    #include "SIO4DocSrcLib.h"
```

```
int sio4_async_setup_tx(
    int         fd,
    CLOCK_RATE  rate,
    STOP_BITS   bits,
    int         verbose)
{
    XMT_ASYNC_PROTOCOL  parm;
    int                 status;

    parm.eTxClockRate   = rate;
    parm.eTxStopBits    = bits;
    status  = ioctl(fd, SIO4_SET_XMT_ASYNC_PROT, &parm);

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

    return(status);
}
```

3.4.67. SIO4_SET_XMT_HDLC_PROT

This service configures the transmitter specific HDLC/SDLC parameters.

Usage

`ioctl()` **Argument**	**Description**
`request`	`SIO4_SET_XMT_HDLC_PROT`
`arg`	`XMT_HDLC_PROTOCOL*`

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_hdlc_setup_tx(int fd, int verbose)
{
    XMT_HDLC_PROTOCOL   parm;
    int                 status;

    parm.u8SharedZeroFlags  = 0;
    parm.u8TxPreambleEnable = 0;
    parm.eTxUnderrun        = ABORT_COND;
    status  = ioctl(fd, SIO4_SET_XMT_HDLC_PROT, &parm);

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

    return(status);
}
```

3.4.68. SIO4_SET_XMT_HDLC_SDLC_LOOP_PROT

This service configures the transmitter specific HDLC/SDLC Loop parameters.

Usage

`ioctl()` Argument	Description
`request`	`SIO4_SET_XMT_HDLC_SDLC_LOOP_PROT`
`arg`	`XMT_HDLC_SDLC_LOOP_PROTOCOL*`

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_hdlc_sdlc_loop_setup_rx(int fd, int verbose)
{
    XMT_HDLC_SDLC_LOOP_PROTOCOL parm;
    int                         status;

    parm.u8SharedZeroFlags  = 1;
    parm.u8TxActiveOnPoll   = 0;
    parm.eTxUnderrun        = ABORT_COND;
    status  = ioctl(fd, SIO4_SET_XMT_HDLC_SDLC_LOOP_PROT, &parm);

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

    return(status);
}
```

3.4.69. SIO4_SET_XMT_ISOCHR_PROT

This service configures the transmitter specific Isochronous parameters.

Usage

`ioctl()` Argument	Description
`request`	`SIO4_SET_XMT_ISOCHR_PROT`
`arg`	`XMT_ISOCHR_PROTOCOL*`

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_isochr_setup_tx(int fd, UINT8 two_stop_bits, int verbose)
```

```
{
    XMT_ISOCHR_PROTOCOL parm;
    int                 status;

    parm.u8TwoStopBits  = two_stop_bits;
    status  = ioctl(fd, SIO4_SET_XMT_ISOCHR_PROT, &parm);

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

    return(status);
}
```

3.4.70. SIO4_TX_CABLE_CLOCK_CONFIG

This service configures the channel's use of the Tx Cable Clock signal and retrieves its current configuration. If one of the predefined configurations is requested, it is applied. If the corresponding `XXX_READ` macro is supplied, then the current configuration is not changed. Before returning, the current configuration is obtained and reported to the caller. If the feature is unsupported, then the corresponding `XXX_UNKNOWN` macro is returned. If the feature is supported but the configuration is invalid, then the corresponding `XXX_INVALID` macro is returned.

Usage

`ioctl()` Argument	**Description**
`request`	`SIO4_TX_CABLE_CLOCK_CONFIG`
`arg`	`__s32*`

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_tx_cable_clock_config(
    int     fd,
    __s32   set,
    __s32*  get,
    int     verbose)
{
    int status;

    status  = ioctl(fd, SIO4_TX_CABLE_CLOCK_CONFIG, &set);

    if (get)
        get[0]  = set;

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

    return(status);
}
```

3.4.71. SIO4_TX_CABLE_CONFIG

This service configures the transmitter's connection to the cable interface and retrieves its current configuration. If one of the predefined configurations is requested, it is applied. If the corresponding `XXX_READ` macro is supplied, then the current configuration is not changed. Before returning, the current configuration is obtained and reported to the caller. If the configuration is invalid, then the corresponding `XXX_INVALID` macro is returned.

Usage

`ioctl()` Argument	**Description**
`request`	`SIO4_TX_CABLE_CONFIG`
`arg`	`__s32*`

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_tx_cable_config(int fd, __s32 set, __s32* get, int verbose)
{
    int status;

    status  = ioctl(fd, SIO4_TX_CABLE_CONFIG, &set);

    if (get)
        get[0]  = set;

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

    return(status);
}
```

3.4.72. SIO4_TX_CABLE_DATA_CONFIG

This service configures the channel's use of the Tx Cable Data signal and retrieves its current configuration. If one of the predefined configurations is requested, it is applied. If the corresponding `XXX_READ` macro is supplied, then the current configuration is not changed. Before returning, the current configuration is obtained and reported to the caller. If the feature is unsupported, then the corresponding `XXX_UNKNOWN` macro is returned. If the feature is supported but the configuration is invalid, then the corresponding `XXX_INVALID` macro is returned.

Usage

`ioctl()` Argument	**Description**
`request`	`SIO4_TX_CABLE_DATA_CONFIG`
`arg`	`__s32*` (see page 33 for valid values)

Example

```
#include <errno.h>
#include <stdio.h>
```

```
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_tx_cable_data_config(
    int     fd,
    __s32   set,
    __s32*  get,
    int     verbose)
{
    int status;

    status  = ioctl(fd, SIO4_TX_CABLE_DATA_CONFIG, &set);

    if (get)
        get[0]  = set;

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

    return(status);
}
```

3.4.73. SIO4_TX_FIFO_AE_CONFIG

This service configures the Tx FIFO Almost Empty level and reports the current level. When applying a setting, the Tx FIFO is reset and the current content is lost. If the corresponding `XXX_READ` macro is supplied then no change is applied. Before returning the current programmed level is obtained and supplied to the caller.

Usage

`ioctl()` Argument	**Description**
request	SIO4_TX_FIFO_AE_CONFIG
arg	__s32*

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_tx_fifo_ae_config(int fd, __s32 set, __s32* get, int verbose)
{
    int status;

    status  = ioctl(fd, SIO4_TX_FIFO_AE_CONFIG, &set);

    if (get)
        get[0]  = set;

    if ((verbose) && (status == -1))
        printf("ioctl() failure, errno = %d\n", errno);
    return(status);
}
```

3.4.74. SIO4_TX_FIFO_AF_CONFIG

This service configures the Tx FIFO Almost Full level and reports the current level. When applying a setting, the Tx FIFO is reset and the current content is lost. If the corresponding `XXX_READ` macro is supplied then no change is applied. Before returning the current programmed level is obtained and supplied to the caller.

Usage

`ioctl()` Argument	Description
request	SIO4_TX_FIFO_AF_CONFIG
arg	__s32*

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_tx_fifo_af_config(int fd, __s32 set, __s32* get, int verbose)
{
    int status;

    status  = ioctl(fd, SIO4_TX_FIFO_AF_CONFIG, &set);

    if (get)
        get[0]  = set;

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

    return(status);
}
```

3.4.75. SIO4_TX_FIFO_COUNT

This service retrieves the current Tx FIFO fill level. The value obtained is either the number of bytes of data in the Tx FIFO or the corresponding `XXX_UNKNOWN` macro if the Tx FIFO Count Register is unsupported.

Usage

`ioctl()` Argument	Description
request	SIO4_TX_FIFO_COUNT
arg	__s32*

Example

```
#include <errno.h>
#include <stdio.h>
```

```
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_tx_fifo_count(int fd, __s32* get, int verbose)
{
    int status;

    status  = ioctl(fd, SIO4_TX_FIFO_COUNT, get);

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

    return(status);
}
```

3.4.76. SIO4_TX_FIFO_SIZE

This service retrieves the size of the Tx FIFO. The value obtained is either the capacity of the Tx FIFO in bytes or the corresponding `XXX_UNKNOWN` macro if the Tx FIFO Size Register is unsupported.

Usage

`ioctl()` Argument	**Description**
`request`	`SIO4_TX_FIFO_SIZE`
`arg`	`__s32*`

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_tx_fifo_size(int fd, __s32* get, int verbose)
{
    int status;

    status  = ioctl(fd, SIO4_TX_FIFO_SIZE, get);

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

    return(status);
}
```

3.4.77. SIO4_TX_FIFO_TYPE

This service reports the Tx FIFO’s firmware/hardware type implementation. The value obtained is either the FIFO’s type or the corresponding `XXX_UNKNOWN` macro if the type cannot be determined.

Usage

`ioctl()` **Argument**	**Description**
`request`	`SIO4_TX_FIFO_TYPE`
`arg`	`__s32*`

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_tx_fifo_type(int fd, __s32* get, int verbose)
{
    int status;

    status  = ioctl(fd, SIO4_TX_FIFO_TYPE, get);

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

    return(status);
}
```

3.4.78. SIO4_TX_IO_ABORT

This service aborts an active data write operation (an active `write()` request). This has no affect if no such request is active and it has no affect on future requests.

Usage

`ioctl()` **Argument**	**Description**
`request`	`SIO4_TX_IO_ABORT`
`arg`	Unused

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_tx_io_abort(int fd, int verbose)
{
    int status;

    status  = ioctl(fd, SIO4_TX_IO_ABORT);

    if ((verbose) && (status == -1))
        printf("ioctl() failure, errno = %d\n", errno);
    return(status);
}
```

3.4.79. SIO4_TX_IO_MODE_CONFIG

This service updates and reports the mode used by the driver for data write operations. This refers to how data is moved from host memory to the SIO4 when the `write()` function is called.

Usage

`ioctl()` Argument	Description
request	SIO4_TX_IO_MODE_CONFIG
arg	__s32*

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_tx_io_mode_config(int fd, __s32 set, __s32* get, int verbose)
{
    int status;

    status  = ioctl(fd, SIO4_TX_IO_MODE_CONFIG, &set);

    if (get)
        get[0]  = set;

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

    return(status);
}
```

3.4.80. SIO4_TXC_USC_CONFIG

This service configures the channel’s use of the USC TxC signal and retrieves its current configuration. If one of the predefined configurations is requested, it is applied. If the corresponding `XXX_READ` macro is supplied, then the current configuration is not changed. Before returning, the current configuration is obtained and reported to the caller. If the feature is unsupported, then the corresponding `XXX_UNKNOWN` macro is returned. If the feature is supported but the configuration is invalid, then the corresponding `XXX_INVALID` macro is returned.

Usage

`ioctl()` Argument	Description
request	SIO4_TXC_USC_CONFIG
arg	__s32*

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_txc_usc_config(int fd, __s32 set, __s32* get, int verbose)
{
    int status;

    status  = ioctl(fd, SIO4_TXC_USC_CONFIG, &set);

    if (get)
        get[0]  = set;

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

    return(status);
}
```

3.4.81. SIO4_WRITE_REGISTER

This service writes a value to an SIO4 register. This includes GSC firmware and USC registers only. All PCI and PLX feature set registers are read-only. Refer to the SIO4 User Manual and to `sio4.h` for a complete list of available registers. Applications should exercise care in writing to some of these registers. This is because some are used by the driver for interrupt and DMA purposes. Writing to these registers may interfere with proper SIO4 and driver operation and may disrupt the stability of the operating system. The registers of concern are those listed below.

- The GSC Board Control Register
- The GSC Interrupt Control Register (and the interrupt configuration registers)
- The GSC Interrupt Status Register
- The USC Bus Configuration Register
- The USC Daisy Chain Control Register
- The USC Interrupt Control Register

WARNING: Writing to some registers may interfere with proper driver operation and may potentially disrupt the stability of the operating system.

Usage

`ioctl()` Argument	Description
request	SIO4_WRITE_REGISTER
arg	REGISTER_PARAMS*

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

int sio4_reg_write(int fd, __u32 reg, __u32 value, int verbose)
{
    REGISTER_PARAMS parm;
    int             status;

    parm.u32RegisterNumber  = reg;
    parm.u32Value           = value;
    status  = ioctl(fd, SIO4_WRITE_REGISTER, &parm);

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

    return(status);
}
```

4. Operation

This section explains some operational procedures on using the driver. This is in no way intended to be a comprehensive guide on using the SIO4 and makes no attempt at explaining configuration of the Zilog Z16C30. This is simply to address a very few issues relating to GSC specific features of the SIO4.

4.1. Signal Routing

One of the basic requirements for proper operation of the SIO4 is defining how various signals are to be used. This section gives on overview of the SIO4's signal routing options, including references to the applicable driver services. On newer SIO4s signal routing is controlled by firmware only. On older boards signal routing also requires manual adjustment of on-board jumpers. All listed driver services apply all register modifications needed to configure the respective routing option. This includes configuration of pertinent USC and GSC firmware registers. This section does not otherwise pertain to signal routing inside the USC. The figure below gives an overall picture of the board's signal routing features. Each block in the figure represents one or more configurable features which are further described in subsequent paragraphs (except for the cable connector block).

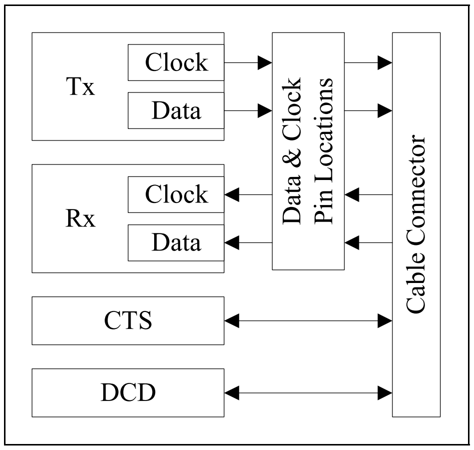

Figure 1 An overview of the SIO4 signal routing features.

4.1.1. Data and Clock Cable Pin Locations

The SIO4 permits the location of the data and clock signals to be interchanged. The default is that the signals are disabled. The cable locations where these signals may appear are referred to as upper and lower in the hardware manual. When the enabled, the two Tx signals are always outputs and the two Rx signals are always inputs. The table below identifies the driver services used to configure routing of the data signals.

Signal	Description	Driver Service
Tx/Rx	These can be configured in most any combination of disabled, lower and upper.	`SIO4_CABLE_CONFIG` (IOCTL page 53, options page 28)

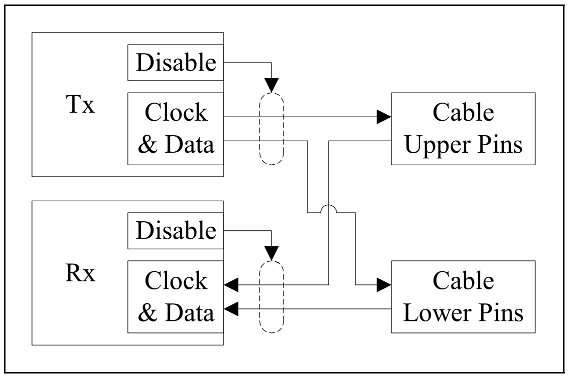

Figure 2 Cable pin location options for the data and clock signals.

4.1.2. Tx and Rx Clocks

The SIO4 includes a Tx Clock cable signal (TxClk) and an Rx Clock cable signal (RxClk), though they are not always used. TxClk is always an output and RxClk is always an input. The table below identifies the driver services used to configure routing of the clock signals.

Signal	Description	Driver Service
RxClk	This is not configurable.	None.
TxC	This can be configured to input any of the listed sources. It can also be configured to output an internal USC clock source.	`SIO4_TXC_USC_CONFIG` (IOCTL page 105, options page 33)
RxC	This can be configured to input any of the listed sources. It can also be configured to output an internal USC clock source.	`SIO4_RXC_USC_CONFIG` (IOCTL page 83, options page 32)
TxClk	This can be configured to output any of the listed sources.	`SIO4_TX_CABLE_CLOCK_CONFIG` (IOCTL page 99, options page 33)

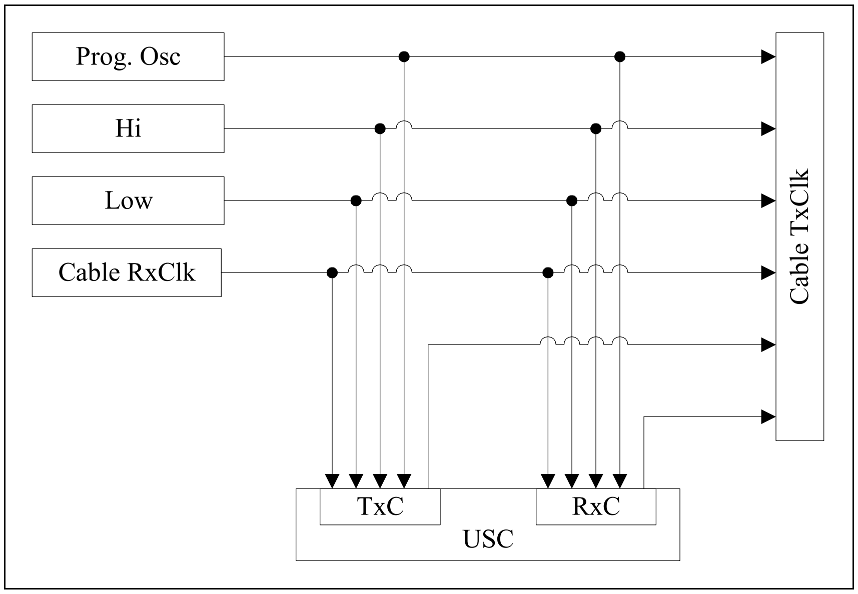

Figure 3 Cable clock signal routing options.

4.1.3. Tx and Rx Data

The SIO4 includes a Tx Data cable signal (TxD) and an Rx Data cable signal (RxD), though both are not always used. TxD is always an output and RxD is always an input. The table below identifies the driver services used to configure routing of the data signals.

Signal	Description	Driver Service
RxD	This is not configurable.	None.
TxD	This can be configured to output any of the listed sources.	`SIO4_TX_CABLE_DATA_CONFIG` (IOCTL page 100, options page 33)

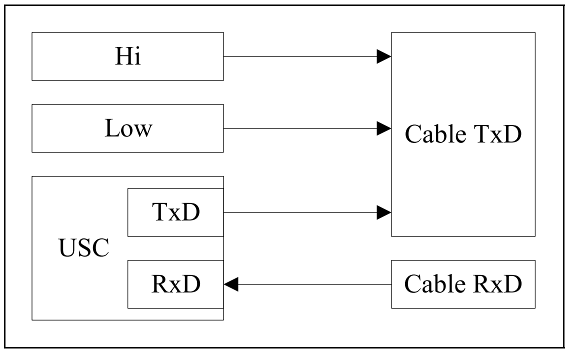

Figure 4 Cable data signal routing options.

4.1.4. CTS

The SIO4 includes a CTS cable signal (CTS), though it is not always used. The signal may be configured for multiple operating modes as either an input or an output. The table below identifies the driver services used to configure routing of the CTS signal.

Signal	Description	Driver Service
CTS	This can be configured to function in any of the listed modes.	`SIO4_CTS_CABLE_CONFIG` (IOCTL page 54, options page 29)

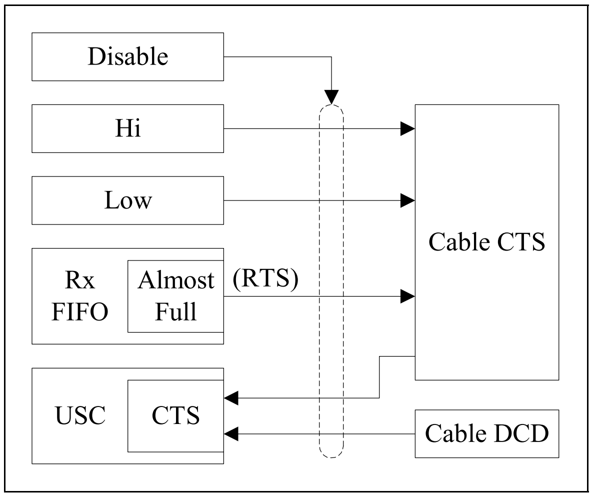

Figure 5 Cable CTS signal routing options.

4.1.5. DCD

The SIO4 includes a DCD cable signal (DCD), though it is not always used. The signal may be configured for multiple operating modes as either an input or an output. The table below identifies the driver services used to configure routing of the DCD signal.

Signal	Description	Driver Service
DCD	This can be configured to function in any of the listed modes.	`SIO4_DCD_CABLE_CONFIG` (IOCTL page 55, options page 29)

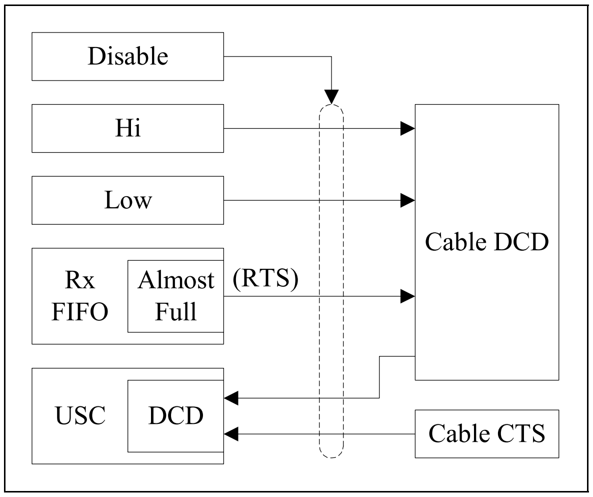

Figure 6 Cable DCD signal routing options.

4.2. I/O Modes

The following describes the three supported I/O modes used for data transfer between the host and the SIO4. All three modes are available using the C library routines `read()` and `write()`. Applications select the desired mode using IOCTL services. Use the `SIO4_TX_IO_MODE_CONFIG` IOCTL service to configure the `write()` data transfer mode and use the `SIO4_RX_IO_MODE_CONFIG` IOCTL service to configure the `read()` data transfer mode.

4.2.1. DMDMA

This refers to Demand Mode DMA. This mode transfers data with the least amount of CPU overhead. It accommodates transfers that exceed the size of the installed FIFOs and uses the FIFO fill level to throttle data movement over the PCI bus. This permits efficient data movement over the PCI bus and also permits the transfer to remain active while data is being transferred over the cable interface. Since the SIO4 can have up to eight data streams (4 Rx and 4 Rx) and only two DMA engines are available, applications must make selective use of DMA and non-DMA I/O requests. Applications can make DMDMA mode I/O requests without having to monitor FIFO fill levels.

4.2.2. DMA

This refers to Non-Demand Mode DMA. This mode transfers data with little CPU overhead, but is suitable only for requests that do not exceed the size of the installed FIFOs. Using this mode, applications much monitor a FIFO's fill level to insure that it can accommodate desired requests. Calling `read()` when the Rx FIFO contains insufficient data will result in indeterminate data at the point where the FIFO runs empty. Calling `write()` when the Tx FIFO contains insufficient free space will result in data loss at the point the FIFO becomes full. Since the SIO4 can have up to eight data streams (4 Rx and 4 Rx) and only two DMA engines are available, applications must make selective use of DMA and non-DMA I/O requests.

4.2.3. PIO

This mode uses repetitive register accesses. While it is the least efficient method it accommodates simultaneous transfers on any number of channels and in both directions. Applications can make PIO mode I/O requests without having to monitor FIFO fill levels.

4.3. Onboard DMA

The SIO4 is designed to automatically transfer data between the USC and the channel's FIFOs. This is done using DMA, which is a feature built-in to the USC and supported by SIO4 circuitry. This feature can be configured by invoking the `SIO4_SET_USC_DMA_OPTIONS` IOCTL service. Doing this manually requires that register fields be set as follows.

Register	Setting
`USC.IOCR.TxRMode`	`1`
`USC.IOCR.RxRMode`	`1`
`USC.HCR.TxAMode`	`1`
`USC.HCR.RxAMode`	`1`

4.4. Oscillator Programming

The ability to program the SIO4's onboard oscillators depends on the board's hardware capabilities and on support included in the driver. The driver can identify the oscillator chip for all SIO4 implementations up to and including those using the Cypress CY22393 Programmable Oscillator. At present however, the driver includes built-in programming support only for those SIO4s using the CY22393. The driver will return an error status when exercising the programmable oscillator features for all other programmable oscillator types. The general procedure to follow when using the programmable oscillator features are as follows.

> **NOTE:** The driver measures the SIO4's reference frequency when the driver is first loaded. If it cannot be measures, then it is initialized to 20MHz. Thereafter, the reference frequency is changed only when done explicitly by application requests using the `SIO4_OSC_REFERENCE` IOCTL service.

1. Determine if the driver is able to perform oscillator programming for the device. This can be done using the `SIO4_FEATURE_TEST` IOCTL service on the `SIO4_FEATURE_OSC_PROGRAM` feature. If the feature in unsupported, then do not attempt programming. Attempting to use the driver's built-in programming features will be unsuccessful when this feature is unsupported. If the feature is supported, then continue with the following steps.

2. Tell the driver the SIO4's reference frequency. This is done using the `SIO4_OSC_REFERENCE` IOCTL service. The specified reference frequency is applicable to all channels since the SIO4 has only a single reference oscillator. The specified reference frequency is used for subsequent operations only.

3. Reset the channel's clock. This is done using the `SIO4_OSC_RESET` IOCTL service. Depending on the oscillator, this may disable the channel's clock. Depending on the SIO4, this effort may affect all channels.

4. Initialize the channel's clock. This is done using the `SIO4_OSC_INIT` IOCTL service. Depending on the oscillator, this should configure the channel to output the reference frequency. Depending on the SIO4, this effort may affect all channels.

5. Request that the oscillator be reprogrammed for the desired frequency. This is done using the `SIO4_OSC_PROGRAM` IOCTL service. The resulting frequency will be as close as possible to the requested frequency. How close this actually is depends on the oscillator's capabilities, its current resource usage and the reference frequency. Check the `sio4_osc_t` structure's `freq_got` field after programming to verify that the resulting frequency is sufficient. Depending on the SIO4, the programming effort may affect all channels.

   NOTE: On occasion, the oscillator programming effort may not take full affect even though the operation completes successfully. Applications should therefore measure the oscillator frequency following programming requests. If the measured results differ significantly from what the programming request indicated would be produced, then repeat the programming and measurement steps until the results are satisfactory.

6. If desired, the channel's current frequency can be measured at any time using the `SIO4_OSC_MEASURE` IOCTL service. However, this should only be done if the frequency can be measured. This capability depends on the SIO4's feature set. Support for this feature can be determined by using the `SIO4_FEATURE_TEST` IOCTL service with the `SIO4_FEATURE_OSC_MEASURE` feature argument.

7. If desired, the current configuration may be determined at any time using the `SIO4_OSC_INFO` IOCTL service. The information returned will be based on the driver's recorded state information.

4.4.1. Cypress CY22393 Programmable Oscillator Support

The SIO4's support for this device includes a fixed reference oscillator, a Cypress CY22393 (with four programmable oscillators), and four firmware based post dividers. The driver defaults the reference frequency to the measured frequency at startup and initializes the programmable oscillators to their off state. The driver manages the firmware post dividers and the CY22393, with its oscillators and Digital Phase Lock Loop Generators, as best as possible to fulfill application requests. The driver responds to the services according to the following table.

Service	**Response**
`SIO4_OSC_INFO`	The current settings are reported.
`SIO4_OSC_INIT`	The desired frequency is set to the reference frequency and the channel is reconfigured accordingly.
`SIO4_OSC_MEASURE`	The output frequency is measured using SIO4 firmware resources. The measured value is reported in the `freq_got` field.
`SIO4_OSC_PROGRAM`	If the requested frequency is non-negative and 20MHz or less, then the driver programs in that configuration that will most closely match the request. This is done based on the CY22393's resources available at that moment.
`SIO4_OSC_REFERENCE`	The requested value is recorded if it is 8MHz or higher and 30MHz or lower.
`SIO4_OSC_RESET`	The desired frequency is set to zero and the channel is reconfigured accordingly.

4.4.2. Cypress IDC2053B Programmable Oscillator Support

The driver does not include support for this device. The driver returns EIO for all programmable oscillator requests when the SIO4 uses this chip.

4.4.3. Fixed Oscillator Support

When the SIO4 has a fixed oscillator, no programming can be performed. Rather than return errors though, the driver treats the hardware as a programmable oscillator capable only of supply the reference frequency. The driver responds to the IOCTL services according to the following table.

Service	**Response**
`SIO4_OSC_INFO`	The current settings are reported.

`SIO4_OSC_INIT`	The `freq_got` value is updated to the reference frequency.
`SIO4_OSC_MEASURE`	The `freq_got` value is reported as `-1` (due to firmware limitations).
`SIO4_OSC_PROGRAM`	The requested value is recorded if it is non-zero and 20MHz or lower.
`SIO4_OSC_REFERENCE`	The requested value is recorded if it is 1MHz or higher and 20MHz or lower.
`SIO4_OSC_RESET`	The `freq_got` value is updated to the reference frequency.

4.4.4. All Other Cases

This applies when the SIO4 includes no programmable oscillator support and when the SIO4 uses a programmable oscillator unrecognized by the driver. The driver responds to the IOCTL services according to the following table.

Service	**Response**
`SIO4_OSC_INFO`	The current recorded settings are reported.
`SIO4_OSC_INIT`	The recorded `freq_want` and `freq_got` values are set to the reference frequency.
`SIO4_OSC_MEASURE`	The `freq_got` value is reported as zero.
`SIO4_OSC_PROGRAM`	The recorded `freq_want` and `freq_got` values are set to the requested value if it is non-zero and 20MHz or lower.
`SIO4_OSC_REFERENCE`	The requested value is recorded if it is 1MHz or higher and 20MHz or lower.
`SIO4_OSC_RESET`	The recorded `freq_want` and `freq_got` values are set to zero.

4.5. Multi-Protocol Transceiver Programming

This feature includes boards with varying capabilities. Some boards are able to change the transceiver protocol under software control. Some have fixed transceiver protocols and can report the protocol via firmware. Others have fixed transceiver protocols, but are not able to report the protocol. The general procedure to follow when using this feature is as follows.

1. Determine if the SIO4 supports this feature. This can be done using the `SIO4_FEATURE_TEST` IOCTL service on the `SIO4_FEATURE_MP` feature. If this feature in unsupported, then do not attempt to exercise the board's Multi-Protocol transceiver feature. Attempting to do so will be unsuccessful when this feature is unsupported. If the feature is supported, then continue with the following steps.

2. Determine if the SIO4's transceiver protocol can be changed. This can be done using the `SIO4_FEATURE_TEST` IOCTL service on the `SIO4_FEATURE_MP_CHANGE` feature. If this feature is unsupported, then do not attempt to exercise the board's Multi-Protocol transceiver feature. Attempting to do so will be unsuccessful when this feature is unsupported. If the feature is supported, then continue with the following steps.

3. Determine if the transceiver protocol desired is supported. This can be done using the `SIO4_MP_TEST` IOCTL. If a suitable protocol cannot be selected, then do not attempt to further exercise the board's Multi-Protocol transceiver feature. If a suitable protocol is available, then continue with the following steps.

4. Select a suitable transceiver protocol. This can be done using the `SIO4_MP_CONFIG` IOCTL.

5. If desired, the current configuration can be determined at any time using the `SIO4_OSC_INFO` IOCTL service.

4.5.1. Cipex SP508 Multi-Protocol Transceiver Support

When the SIO4 includes these transceiver chips, the driver responds to the services according to the following table.

Service	Response
`SIO4_MP_CONFIG`	The chip will be given as the SP508 option. The resulting protocol will equal the requested protocol if it is supported. The resulting protocol will otherwise be the invalid option.
`SIO4_MP_INFO`	The chip will be given as the SP508 option. The desired protocol will be the read option. The resulting protocol will reflect the board's current configuration.
`SIO4_MP_INIT`	The chip will be given as the SP508 option. The desired and resulting protocol will both be the RS-422/485 option.
`SIO4_MP_RESET`	The chip will be given as the SP508 option. The desired and resulting protocol will both be the disable option.
`SIO4_MP_TEST`	The chip will be given as the SP508 option. The resulting protocol will be the requested protocol if it is supported. The resulting protocol will otherwise be the invalid option.

4.5.2. Fixed Protocol Support

Some SIO4s include Multi-Protocol support in firmware but not in hardware. This applies when the SIO4 has fixed transceivers whose type is reported by firmware. Under these circumstances the driver responds to the IOCTL services according to the following table.

Service	Response
`SIO4_MP_CONFIG`	The chip will be given as the fixed option. The resulting protocol will reflect the board's hardwired protocol.
`SIO4_MP_INFO`	The chip will be given as the fixed option. The desired protocol will be the read option and the resulting protocol will reflect the board's hardwired protocol.
`SIO4_MP_INIT`	The chip will be given as the fixed option. The desired and resulting protocols will reflect the board's hardwired protocol option.
`SIO4_MP_RESET`	The chip will be given as the fixed option. The desired and resulting protocols will reflect the board's hardwired protocol.
`SIO4_MP_TEST`	The chip will be given as the fixed option. The resulting protocol will be the test protocol if it is the board's hardwired protocol. The resulting protocol will otherwise be the invalid option.

4.5.3. All Other Cases

This applies when the firmware includes no Multi-Transceiver protocol support and when support is present but the protocol is fixed. In these cases the driver responds to the IOCTL services according to the following table.

Service	Response
`SIO4_MP_CONFIG`	The chip and resulting protocol will each be given as their respective unknown options.
`SIO4_MP_INFO`	The desired protocol will be the read option. The chip and resulting protocol will each be given as their respective unknown options.
`SIO4_MP_INIT`	The chip, the desired protocol and resulting protocol will all be given as their respective unknown options.
`SIO4_MP_RESET`	The desired protocol will be the disable option. The chip and resulting protocol will each be given as their respective unknown options.
`SIO4_MP_TEST`	The chip and resulting protocol will each be given as their respective unknown options.

4.6. Interrupt Notification

Applications can make indirect use of SIO4 interrupts by using the Interrupt Notification IOCTL services. This requires the following basic steps. These steps are illustrated in the source code sample that follows.

1. Use the `fcntl` interface to register the application's signal handler.

2. If USC interrupts are to be used, then configure the USC for the interrupts desired. Consult the Zilog data book for the required register settings.

3. Issue the `SIO4_INT_NOTIFY` IOCTL service to request notification.

4. When the SIGIO signal is received, issue the `SIO4_READ_INT_STATUS` IOCTL service to determine which interrupt occurred. If a USC interrupt was received then examine the USC to determine which interrupt occurred and clear it.

WARNING: If a USC interrupt occurs then that interrupt must be serviced within the USC by the application. If this is not done then that interrupt source within the USC will continue to function as an active USC interrupt source. In this case the SIO4 will continue to assert an interrupt while USC interrupts are enabled.

5. Perform any application required actions.

6. If additional notification is required for an interrupt that was reported then repeat steps two through five as required.

7. When finished issue the `SIO4_INT_NOTIFY` IOCTL service with an argument value of zero (0) to specify that notification be terminated.

Example

```
#include <errno.h>
#include <fcntl.h>
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "SIO4DocSrcLib.h"

static int  _fd;

static void handle_sigio(int signo)
{
    SIO4_INTERRUPT_STATUS   int_stat;
    int                     status;

    status  = ioctl(_fd, SIO4_READ_INT_STATUS, &int_stat);

    if (status == -1)
    {
        /* The request failed. */
    }
    else if (int_stat.u8SIO4Status & SIO4_INT_NOTIFY_TX_FIFO_AE)
    {
        /* Handle the Tx FIFO Almost Empty condition. */
    }
}

int sio4_async_setup(int fd)
{
    int             flags;
    unsigned char   notify;
    pid_t           pid;
    int             status;

    ioctl(fd, SIO4_INT_NOTIFY, 0);
    _fd = fd;
    signal(SIGIO, handle_sigio);
    pid = getpid();
    fcntl(fd, F_SETOWN, pid);
    flags   = fcntl(fd, F_GETFL);
    flags   |= FASYNC;
    fcntl(fd, F_SETFL, flags);
    notify  = SIO4_INT_NOTIFY_TX_FIFO_AE;
    status  = ioctl(fd, SIO4_INT_NOTIFY, notify);
    return(status);
}
```

Document History

Revision	Description
September 30, 2005	Updated to release 1.18.2.
September 26, 2005	Updated to release 1.18.1.
July 15, 2005	Updated to release 1.18.0. Removed feature definitions that are no longer supported.
May 24, 2005	Updated to release 1.17.1.
May 19, 2005	Updated to release 1.17.0.
May 10, 2005	Updated to release 1.16.0. Corrected timeout information. Added new feature options.
April 5, 2005	Updated to release 1.15.1.
March 23, 2005	Updated to release 1.15.0.
January 25, 2005	Updated to release 1.14.0. Updated the driver to support the 2.6 kernel.
November 3, 2004	Updated to release version 1.12.1.
November 2, 2004	Updated to release version 1.12.0. Added operation information on signal routing options. Added the IOCTL service `SIO4_TX_CABLE_DATA_CONFIG`. Expanded the set of valid values for the IOCTL service `SIO4_CTS_CABLE_CONFIG`. Added the `SIO4_DCD_CABLE_CONFIG` IOCTL service. Added the `SIO4_CABLE_CONFIG` IOCTL service.
October 18, 2004	Updated to release version 1.11.0. Updated interrupt notification sample code. Added a sample application, asyncc2c, which performs asynchronous channel-to-channel data transfers. Removed the `SIO4_RX_CABLE_CLOCK_CONFIG` IOCTL service as it isn't in firmware.
August 30, 2004	Updated to release version 1.10.0.
August 18, 2004	Updated to release version 1.09.0. Updated documentation on some init and reset services.
August 17, 2004	Updated to release version 1.08.0. Fixed driver `SIO4_INIT_CHANNEL` bug.
August 11, 2004	Updated to release version 1.07.2. Changed UART references to USC.
August 10, 2004	Updated to release version 1.07.1. Removed PMC-SIO4AR from front page as some device features are not properly supported on this board.
August 9, 2004	Updated to release version 1.07.0. Added PMC-SIO4AR to front page.
July 28, 2004	Updated the list of SIO4 models covered by this user manual. Added the IOCTL service `SIO4_MOD_REGISTER` and the data structure `REGISTER_MOD_PARAMS`. Added the `SIO4_READ_REGISTER_RAW` IOCTL service. Updated numerous register names. Added new feature test options. Added programmable oscillator IOCTL services and a support data structure. Added Multi-Protocol Transceiver IOCTL services and support data structures. Updated the archive directory structure and reorganized the relevant document sections. Reversed the history list to show newest changes first. Removed the DMA IOCTL services. Expanded `read()` and `write()` to use DMA and DMDMA. Added the I/O Mode Configuration IOCTL services. Added the I/O Abort services. Corrected bugs in the `SIO4_RESET_FIFO` and `SIO4_SEND_CHANNEL_COMMAND` code samples.
March 23, 2004	Added services and updated example code. Updated numerous register macros. Only firmware and USC registers are writable. The PCI and PLX registers are now read-only. The document source code samples are now provided as a library.
March 1, 2004	Removed the "tainting" remarks as the driver is now covered by GPL.
April 29, 2003	Added more registers and did additional porting.
November 19, 2002	More porting, bug fixing and minor corrections.
August 5, 2002	Ported the driver to the 2.4 kernel.
June 25, 2002	Minor correction.
January 29, 2002	Initial release.