SIO4/8

Four/Eight Channel High Speed Serial I/O

All SIO4 and SIO8 Models
All Form Factors
All Standard Zilog Versions
All Standard SYNC Versions

Porting Guide From The GSCAPI To The 3.x Series Monolithic Driver

Manual Revision: March 19, 2024

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

Preface

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

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

General Standards Corporation makes no warranty of any kind with regard to this documentation and/or software, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Although extensive editing and reviews are performed before release, **General Standards Corporation** assumes no responsibility for any errors, inaccuracies or omissions herein. This documentation, information and software are made available solely on an "as-is" basis. Nor is there any commitment to update or keep current this documentation.

General Standards Corporation does not assume any liability arising out of the application or use of documentation, software, product or circuit described herein, nor is any license conveyed under any patent rights or any rights of others.

General Standards Corporation assumes no responsibility for any consequences resulting from omissions or errors in this manual or from the use of information contained herein.

General Standards Corporation reserves the right to make any changes, without notice, to this documentation, software or product, to improve accuracy, clarity, reliability, performance, function, or design.

GSC is a trademark of **General Standards Corporation**.

PLX and PLX Technology are trademarks of PLX Technology, Inc.

Zilog and Z16C30 are trademarks of Zilog, Inc.

Table of Contents

1. Introduction

This guide applies to applications being ported from the GSCAPI interface to the SIO4 API Library interface that is a part of the 3.x series monolithic Linux driver. There have been three SIO4 Linux drivers. The first is a monolithic driver whose last release was version 1.58.93.36.0 dated April 30, 2021. This is now a legacy driver and is not the subject of this porting guide. The second driver is commonly called GSCAPI. This is a library-based driver implementation using a low-level driver produced by PLX Technology. Its last release was version 1.6.10.1 dated March 25, 2020. This legacy driver is the subject of this porting guide. The third driver is the currently maintained SIO4 Linux driver, referred to as the 3.x series monolithic driver. At the time of this writing the last release of this driver was version 3.17.101.44.0 dated December 13, 2022. This is the target driver addressed in this porting guide.

1.1. Purpose

The purpose of this document is to present information that may be useful in porting source code from using the legacy GSCAPI interface to using the actively maintained 3.x series monolithic driver.

1.2. Acronyms

The following is a list of commonly occurring acronyms which may appear throughout this document.

Acronyms	Description
API	Application Programming Interface (This is sometimes used synonymously with API Library.)
DMA	Direct Memory Access
BMDMA	Block Mode DMA (This may refer to non-Scatter-Gather DMA as well as non-Demand Mode DMA.)
DMDMA	Demand Mode DMA
GSC	General Standards Corporation
PIO	Programmed I/O
USC	Universal Serial Controller (This refers to the SIO4’s Zilog Z16C30 serial controller.)

1.3. Definitions

The following is a list of commonly occurring terms which may appear throughout this document.

Term	Definition
…	This is a substitute for a driver’s root installation directory.
API Library	This refers to the library implementing the application-level interface to the monolithic driver.
Application	This refers to user mode processes.
Device Driver	This refers to the driver executable component of the SIO4 driver package.
Driver	This refers to the device driver, which runs under control of the operating system.
Firmware Registers	These are the General Standards specific registers implemented in the board’s FPGA firmware.
Library Driver	This refers to the GSCAPI driver as the SIO4 specific functionality is implemented in a shared object library.
Low Level Driver	This refers to the driver executable included with each driver package.
Monolithic Driver	This refers to the 3.x series driver as virtually all of the SIO4 specific functionality is included in the low-level device driver.
Protocol Library	These are libraries included with the monolithic driver designed to encapsulate support for specific serial protocols. This includes the Asynchronous, HDLC and Isochronous protocols as well as the SYNC protocol used by -SYNC model SIO4s.
SYNC Devices	These are SIO4 devices which use dedicated firmware to implement a synchronous protocol that uses a clock and valid data envelope signal.

Zilog Devices	These are SIO4 devices which use the Zilog Z16C30 USC hardware to support various serial protocols.

2. High Level Issues

This section presents information on high level issues that pertain to other than exercising the driver interfaces.

2.1. Licensing

The two drivers have different licensing restrictions. The GSCAPI's low level driver and its interface library are produced by PLX Technology. Its sources are covered by LGPL version 2 as stated in each of its source files. All remaining content is produced by General Standards and is not explicitly covered by any licensing, unless documented otherwise. The monolithic driver is produced entirely by General Standards. The Linux specific driver sources are covered by GPL. All other sources are released to the general public. Refer to the file `LICENSE.txt` for clarification.

2.2. Kernel Support

Kernel support is the primary reason the GSCAPI is a legacy driver and why the monolithic driver is still being maintained. The GSCAPI supports kernel versions from 2.4 up through 4.9, but no further. The monolithic driver supports kernel versions 2.2 up through 5.x and will be updated further as the need arises. Refer to the respective driver user manuals for clarification.

2.3. Architecture Support

Both drivers support 32-bit x86 and 64-bit x64 architectures. Neither driver has been tested and verified by General Standards on any other architecture. Numerous attempts have been made at testing the monolithic driver on ARM based systems, but all such tests have been unsuccessful. All tests conducted have failed due to BIOS and/or kernel porting deficiencies.

2.4. Programming Language Support

All sources for the drivers are written in the C programming language. Both drivers' primary application header files are usable as-is with both C and C++ applications. Both have also been used with other programming languages, but mostly because those environments contained support for use of C based library interfaces.

2.5. Build Environment

All content for both drivers is written for building by the GNU C compiler suite. This includes compilation, linking, the make utilities and the module build tool. The necessary tools should be available for virtually all Linux distributions. For many distributions these tools are installed by default. For other distributions the tools must be installed after OS installation and setup. Building the monolithic driver executable is done by the module build tool and requires installation of the kernel headers and the kernel development package. These are often installed by default, but may have to be installed manually.

2.6. Model Support

The basic model numbers supported by these drivers are given in the table below. Both include support for multiple models, but not the same models. The basic models supported are given in the table below. The table lists standard Zilog and SYNC board models. The same hardware has been used for custom firmware and is typically designated by additions to the model number. A hypothetical example is PMC-SIO4BX-XXX, where the XXX refers to the custom firmware variation. Refer to the following firmware support section for additional information.

> **NOTE:** The GSCAPI driver also supports a small number of analog I/O boards. Support for these boards is limited. The monolithic driver does not include any support for analog I/O boards.

Base Model	GSCAPI	Monolithic
SIO8BX2	Yes	Yes
SIO8BX2-SYNC	Yes	Yes
SIO8BXS	Yes	Yes
SIO8BXS-SYNC	Yes	Yes
SIO4BX2	Yes	Yes
SIO4BX2-SYNC	Yes	Yes
SIO4BXR	Yes	Yes
SIO4BXR-SYNC	Yes	Yes
SIO4BX	Yes	Yes
SIO4BX-SYNC	Yes	Yes
SIO4B	Yes	Yes
SIO4B-SYNC	Yes	Yes
SIO4AHRM	No	Yes
SIO4AHRM-SYNC	No	Yes
SIO4AR	No	Yes
SIO4AR-SYNC	No	Yes
SIO4A	No	Yes
SIO4A-SYNC	No	Yes
SIO4	No	Yes

2.7. Firmware Support

The SIO4 is a quad channel, bi-directional serial communications board. The standard firmware options include support for the Zilog Z16C30 chips and the -SYNC versions. In addition, numerous firmware variations have been created to implement customer specific protocols. Both drivers have inherent support for much of the custom firmware as the interface changes typically appear as new registers for configuration of the custom protocol. This support only requires making new register definitions, which the custom application uses to configure the custom protocol. The firmware designer goes to great lengths to keep as much common functionality intact as possible to minimize the impact on software. As a result, both drivers continue to function as expected for most, if not all, protocol independent features.

> **NOTE:** The GSCAPI supports a custom firmware version implementing a customer specific protocol called CTC. The monolithic driver does not contain any corresponding content.

2.8. Dynamic Firmware Selection

More recent SIO4 firmware implementations support both the Zilog functionality and -SYNC functionality within the same firmware, presuming the Zilog USC chips are installed. When both are supported, software can switch between the two on a per channel basis by modifying the Firmware Type Register. This feature allows the firmware type to be changed during channel initialization, or reinitialization. This feature is supported by the GSCAPI, but not by the monolithic driver.

> **NOTE:** The firmware type should be changed **only** during initialization and setup.

2.9. Documentation

API documentation is included with each driver. The GSCAPI provides a single PDF document. This manual describes driver installation and provides detailed information on all API function calls and data structures. It also includes Linux man pages. The monolithic driver includes several PDF documents. It includes a Linux specific driver installation manual, an API reference manual plus a reference manual for each of the four Protocol Libraries. The monolithic driver does not include Linux man pages.

2.10. Driver Archive

Both drivers are distributed as compressed archives whose file names include the driver release version number. Each archive includes all of the sources for all build targets as well as the corresponding documentation. The archive file names are given in the table below. The series of X's represents the version number of the release.

Driver	File Name
GSCAPI	`gsc_api.linux.x.x.x.tar.gz`
Monolithic	`sio4.linux.x.x.x.x.x.tar.gz`

2.11. Installation Procedures

The installation procedures for the drivers are similar. The procedures are relatively simple with the basic steps as follows; open a terminal window, create a specified directory then extract the archive content. Consult the corresponding documentation for clarification.

2.12. Installation Directory

The target installation directory for each driver is as noted in the table below. In both cases the default locations are according to convention, though not necessarily conventions that applied at the same point in time. The drivers can however, be installed in locations of the developer's choosing.

Driver	Default Installation Directory	Root Directory
GSCAPI	`/usr/local/`	`…/gsc_api/`
Monolithic	`/usr/src/linux/driver/`	`…/sio4/`

2.13. Directory Structure

The organization of each driver's installed files varies greatly. The GSCAPI file organization is built off the driver's origin, which is PLX Technology. The monolithic driver sources are organized at the highest level according to the function of the sources. The files are generally segregated as driver sources, library sources, include sources, sample sources and protocol sources. Consult the corresponding documentation for clarification.

2.14. Environment Variables

Each driver includes its own set of environment variables, as described in the table below.

Driver	Environment Variable	Description
GSCAPI	`PLX_SDK_DIR`	This defines the root directory for the PLX provided sources.
	`PLX_DEBUG`	If non-zero, the sample application build procedures will produce debug versions of the executables.
Monolithic	`GSC_API_COMP_FLAGS`	This is used to add environment specific flags when compiling the API Library.
	`GSC_API_LINK_FLAGS`	This is used to add environment specific flags when linking the API Library.
	`GSC_LIB_COMP_FLAGS`	This is used to add environment specific flags when compiling the utility libraries, including the Protocol Libraries.
	`GSC_LIB_LINK_FLAGS`	This is used to add environment specific flags when linking the utility libraries, including the Protocol Libraries.
	`GSC_APP_COMP_FLAGS`	This is used to add environment specific flags when compiling the sample applications.
	`GSC_APP_LINK_FLAGS`	This is used to add environment specific flags when linking the sample applications.

2.15. Build Procedures

The two drivers have very different build procedures. The monolithic driver has a single shell script that builds and loads the driver, builds and installs the API Library (a .so file), builds the Protocol Libraries then builds all utility libraries and all sample applications. All targets can be built separately, but the script builds everything with just a single command line. The GSCAPI does not have a corresponding means of building everything. Instead, all targets are built separately. This includes building and loading two versions of the driver, based on which SIO4 models are present, building and installing the interface libraries (.so files) along with a sample application. The other sample applications are built in a similar manner.

2.16. Build Targets: Release vs Debug

Both drivers provide support for generating release and debug versions of build targets. Also, both produce release, non-debug, versions by default. For debug builds the GSCAPI uses the `PLX_DEBUG` environment variable given above (section 2.14, page 11). This pertains to the sample applications, which generates separate executables. It is not clear if this flag is also used by the shared libraries. For the monolithic driver one must perform a clean operation, then request debug builds. The clean operation is requested by adding the argument `clean` to the make command line (or the overall make script). The debug build is requested by adding the argument `debug`. This pertains to all build targets except the device driver. This refers to the API Library shared object file, the Protocol Libraries, the utility libraries and the sample applications. The debug builds produce files with the same name as the release builds. To generate release builds, perform a clean operation followed by a build with either no added argument or with the term `release` added.

2.17. Primary Header Files

Each driver provides a single header file to be included by applications using the respective interface. Via that header file each driver includes one or more other interface specific headers put in place during installation. The files to be included are given in the table below.

Driver	Header File	Default Location
GSCAPI	`GscApi.h`	`/usr/local/include/GscApi/`
Monolithic	`sio4_main.h`	`/usr/src/linux/drivers/sio4/include/`

2.18. Primary Library Files

The drivers implement their primary interfaces as shared object files (.so files). The GSCAPI functionality is split between two different libraries. One is for the GSCAPI published interface and the other is for the PLX API, which is the interface used by the GSCAPI to access the low-level device driver. These libraries are given in the table below. The monolithic driver's library is a thin layer implemented to facilitate porting the driver and applications to other operating systems, which has been done.

Driver	Shard Object File	Default Location
GSCAPI	`libGscApi.so` `libPlxApi.so`	`/usr/local/lib/`
Monolithic	`libsio4_api.so`	`/usr/lib/`

The monolithic driver includes additional utility libraries, including four Protocol Libraries. These are provided via statically linked libraries as given in the table below. The libraries are provided in two flavors. One is for applications using the SIO4 and no other GSC products. The other is for applications which use the SIO4 and one or more other GSC products. This is done because all of the current GSC product Linux drivers' static libraries share common code, which presents a linking error when duplicate symbols are found.

Location	Libraries	Description
`…/lib/`	`sio4_main.a`	Applications link this if they use the SIO4 and no other GSC products.
	`sio4_multi.a`	Applications link this if they use the SIO4 and one or more other GSC products. *

	`gsc_utils.a`
	`plx_utils.a`
	`os_utils.a`
		Applications link these if they use the SIO4 and one or more other GSC products. *

* Refer to the driver user manual for clarification.

2.19. Device Access

Gaining access to SIO4 hardware is somewhat different for each driver. The GSCAPI grants access at the board level. In this interface applications take control of the entire board with all four channels. In making calls, the application specifies the board index to access and, where necessary, the channel index to access. The monolithic driver provides device access at the channel level. In this interface the application takes ownership of one channel at a time without affecting operation of any other channel on the board or any other board. Applications can access any number of channels, but it is done one channel at a time. Exclusivity is another access issue. The GSCAPI does not allow an application to gain exclusive access to an SIO4 board or channel. This driver does not provide a mechanism for gaining exclusive access. The monolithic driver interface allows an application to gain exclusive device access or to grant shared access. Gaining exclusive access or granting shared access is done by way of an argument passed to the open service.

2.20. Device Indexing

The GSCAPI uses one-based indexing while the monolithic driver uses zero-based indexing. The GSCAPI refers to the first board as index one, followed by two, three and so on. Channel indexes for every board are always one, two, three and four. The monolithic driver uses only device indexing, which is equivalent to channel indexing. For the first SIO4 the device index numbers are zero, one, two and three. For the second board device indexes are four, five, six and seven. Indexing continues sequentially thereafter in the same manner.

2.21. Versioning

Both drivers maintain version numbers for their content. The GSCAPI maintains a version number for the API and the low-level device driver. The low-level driver version number is the PLX SDK version number from which the low-level driver is taken. The monolithic driver maintains a single version number based on the device driver content (X.X.X.X.X). The first two numbers are the major and minor revision numbers for the driver's device dependent sources. The third number is the revision level of the device and OS independent driver sources. The fourth number is the revision level of the OS specific driver sources. The last number is the ancillary release number reflecting non-driver source changes to the release.

2.22. Shared Source Code

Each of the two driver releases includes a low-level driver whose source code is used to interface with devices other than the SIO4. This has contributed to making both low level drivers mature and reliable software. The GSCAPI's low level driver is produced by PLX Technology and is made available to all those using PLX produced PCI bridge chips. As a result, this driver, the PLX driver, is used widely by many companies. In addition, it is used by General Standards for virtually all of our Windows drivers. While PLX has been purchased and now goes by a different name, the Linux driver version, which has been around for over 15 years, continues to be maintained and made freely available for use by their customers. The monolithic driver also contains source code used for other than the SIO4. Some of the device driver sources contain functionality that is OS specific and/or device independent. This common code was first released in 2014, was put to wider use in 2015, which is when it was first used by the SIO4 device driver. These sources are now used across virtually all of General Standards' nearly 45 different, currently maintained Linux drivers.

2.23. OS Support

Support for other than Linux is a common aspect of each of these drivers. The GSCAPI was originally developed for Windows and was later migrated to PLX's Linux driver implementation. The Windows version currently supports Windows Server 2019, Windows Server 2016, Windows 10, Windows 8, Windows 7 and even Windows XP. For the GSCAPI, the basic architecture, that is, the set of libraries and the division of the functionality, is the same for both the Linux and the Windows driver releases. The monolithic driver was originally developed under Linux. Its design was intended to aid driver porting to other operating systems, which has occurred. The SIO4 Linux driver has been ported to RTX and to Intime, both of which are real time extensions for Windows. In these cases, the low-level drivers, which contain the exact same functionality as in the SIO4's low level Linux driver, are implemented as real time applications specific to each extension. While the SIO4 Linux driver hasn't been ported to Windows, the common code referenced above has been a part of porting other GSC Linux drivers to the Windows environment. In these cases, the drivers were ported to Windows 10 and used the latest low level PLX Windows driver. Also, the low-level Linux drivers were implemented as Windows DLLs. Should the SIO4 Linux driver be ported to Windows, it too would appear as a Windows DLL.

2.24. Error Reporting

Errors may be encountered by most any software, including these two drivers. In general, each returns zero when calls are successful and an error code when problems are encountered. The error codes reported are different for each interface. The GSCAPI error codes are provided by the low-level driver and are defined in `PlxStat.h`. It is possible the GSCAPI might also return error codes from the system header file `errno.h`, but this may be unlikely. The monolithic driver returned error values are the negative of codes listed in `errno.h`. Positive return values are reported only by I/O services to reflect the number of bytes transferred.

2.25. Data Types

The interface to each driver defines its own set of data types. This includes both intrinsic, basic data types, enumerations, unions and structures. The interfaces do not share any common data type definitions, except by pure coincidence. However, the interfaces do share a number of similar data types, with easily recognized names. The table below lists some shared data types.

GSCAPI	Monolithic	Description
S8	s8	signed 8-bit integer
U8	u8	unsigned 8-bit integer
S16	s16	signed 16-bit integer
U16	u16	unsigned 16-bit integer
S32	s32	signed 32-bit integer
U32	u32	unsigned 32-bit integer
S64	s64	signed 64-bit integer
U64	u64	unsigned 64-bit integer

2.26. Interface Initialization

The GSCAPI does not include a function which must be called to initialize the interface. The monolithic driver does have such a call, which is `sio4_init()`. All other interface functions will return an error status until the API is initialized via this call. Refer to the API reference manual for clarification.

2.27. Service Interface

The methodologies used by the interfaces to access driver functionality are quite different. The GSCAPI mostly provides function calls, each dedicated to a particular functionality. This includes a wide range of functions for tasks large and small, including calls designed for configuring the various supported serial protocols. In contrast, the monolithic driver interface is primarily IOCTL based, in which IOCTL codes and accompanying arguments are passed to the function `sio4_ioctl()` (section 3.2.2, page 19). Most of the IOCTL services are relatively limited in scope. Very few support in-depth capability and even fewer are intended for internal use only. This is similar to the GSCAPI function pair GscSio4SetOption() (section 3.4.9, page 28) and GscSio4GetOption() (section 3.4.10, page 41).

2.28. Serial Protocols

Both drivers provide means of using the full capability of the SIO4 hardware, including its entire complement of serial protocols. However, the level of explicit support for any given protocol varies. The GSCAPI has function calls and data structures intended to offer detailed configuration and use of the Asynchronous protocol, the BiSync and BiSync16 protocols and the HDLC protocol. Each protocol is supported by a set of dedicated function calls with associated data structures. The monolithic driver implements similar support by way of dedicated libraries. These libraries include the Asynchronous Protocol Library, the HDLC Protocol Library, the Isochronous Protocol Library and the SYNC Protocol Library for use with -SYNC boards. Each library includes a protocol specific interface along with dedicated data structures as well as detailed documentation. While both drivers support the Asynchronous and HDLC protocols, their implementations are very different. Refer to each driver’s documentation for clarification.

> **NOTE:** The GSCAPI supports a custom firmware version implementing a customer specific protocol called CTC. The monolithic driver does not contain any corresponding content.

2.29. I/O Operations

The two drivers have entirely different approaches to implementing I/O. The monolithic driver supports read and write calls in which the data transfer mechanism (I/O mode) is a configurable parameter. This driver’s I/O mode options include Programmed I/O (PIO, which is repetitive register accesses), Block Mode DMA (move data after it is known that it can be completed safely) and Demand Mode DMA (start the transfer immediately, but move the data as either Rx FIFO data or Tx FIFO space becomes available). No matter which option is selected all of the footwork is handled by the driver, along with various other configurable parameters. Also, while the DMA engines support Scatter-Gather DMA, the monolithic driver does not. Instead, this driver uses Double Buffering, in which DMA is performed using internal, DMA safe data buffers.

The GSCAPI, on the other hand, has no similar, high level read or write services and no corresponding set of configurable I/O parameters. This interface does not include PIO functionality. Instead, if utilized at all, it is up to the application to perform the repetitive register reads or writes. This driver provides DMA support, but it all has to be managed by the application by choosing the appropriate DMA function.

The footwork mentioned above entails mostly monitoring the FIFO fill levels then performing the data transfer as conditions permit. With the GSCAPI it is up to the application to define and implement these activities. On the other hand, the monolithic driver does all this for the application. This includes checking for overruns and underruns, checking the FIFO fill level, whose methods vary among SIO4 models, examining configured threshold levels, gaining access to a DMA engine, when called for, and, when conditions permit, initiating data transfers using the configured I/O mode. Refer to the API reference manual for clarification.

2.30. DMA Engine Access Policy

Each SIO4 has only two DMA engines that must be shared among a potential of eight I/O threads. Both drivers prudently control access to the DMA engines to help maintain system stability. The GSCAPI policy is to return an error status if a DMA engine is in use when another thread seeks access. As it is the application that specifies which DMA engine to access, it is up to the application to decide how to proceed. Plus, if the application has more than two simultaneous threads that may need DMA engine access, then the application is responsible for implementing a sharing mechanism. The monolithic driver’s policy is to poll for DMA engine availability if both are in use. When this occurs, the driver waits one system timer tick before checking again. The driver will repeat this process until either a DMA engine becomes available or until the configured I/O timeout interval lapses. In this case, the I/O request returns the number of bytes that have transferred, which might be zero. This driver does not treat this situation as an error condition.

2.31. I/O Memory Access

The drivers have different approaches for how I/O memory is accessed and used. The monolithic driver uses double buffering. With this approach the driver allocates a pair of internal, DMA save memory buffers for each channel at the time the driver is loaded. When write requests are made the data is copied from the application buffer to the internal transmit buffer then transferred to the SIO4 transmit FIFO by the preselected method. When read requests are made the data is transferred from the SIO4 receive FIFO to the internal receive buffer by the preselected method, then copied to the application buffer. With this implementation applications do not have direct access to the driver's internal transfer buffers. With the GSCAPI the application is responsible for managing I/O transfer buffers and has direct access to all buffers that are used. The first option is for the application to use its own buffers. This refers to buffers defined as local storage or allocated on the stack. When these buffers are used, the interface initiates Scatter-Gather type DMA to transfer data directly between the SIO4 and these application buffers. No other buffering is performed. However, this method has definite overhead as the memory must be page-locked and uncached (made DMA safe) before the DMA can be performed. Then, afterwards, the memory must be unlocked. The second obtain entails the application going through the driver interface to obtain contiguous, page-locked memory from the low-level driver. While this method requires more work on the part of the application before it can perform I/O, and after it is all done, it avoids the above mentioned overhead associated with every I/O request.

2.32. Oscillator Programming

Oscillator programming is supported under both drivers. With the GSCAPI oscillator programming is performed by the application. With the monolithic driver programming is performed by the device driver. Here, applications use the `SIO4_IOCTL_OSC_PROGRAM` IOCTL service. The service's argument is an `s32*` pointing to the desired frequency.

2.33. Working in the Background

The GSCAPI was created under Windows, whose driver model is notably different from that of Linux. One aspect of this difference is that a lot of Windows driver functionality occurs in the background while the application is free to continue working. A large portion of the GSCAPI interface negates this characteristic by waiting for quick and timely operations to complete before returning control to the application. This does not apply to all DMA operations. The DMA calls initiate the DMA then either wait for completion or return immediately so the application can continue working. The application can wait for completion or it can periodically query for completion status. With the monolithic driver all operations are blocking calls. Most service requests are carried out very quickly and return. Some requests block the caller for a brief period and then return. Still others block the caller, but limit their duration with a configurable timeout period. However, all calls are blocking with no work being carried out in the background.

2.34. Event Notification

The two drivers provide entirely different mechanisms for notifying applications of device and/or driver events. The GSCAPI uses a callback mechanism in which a function is registered to be called when a specified SIO4 firmware interrupt occurs. Each interrupt source is independently configurable, though each interrupt source accommodates only a single callback. The interface doesn't support callback configuration for individual USC interrupts. The monolithic driver uses a Wait Event mechanism in which a thread is blocked until the first of any arbitrary set of events occurs. The implementation accommodates any number of threads waiting on any combinations of events. With this interface, events include all interrupt sources pertaining to the channel as well as I/O call completions. This includes firmware interrupts, USC interrupts and PLX interrupts (i.e., DMA Done interrupts). Refer to the API Library reference manual for clarification.

In addition, the monolithic driver provides event notification for all other interrupts attributable to the board or for which the driver's ISR is called. The most common of these other are for other devices sharing the same interrupt.

2.35. Interrupt Support

Interrupt support is included by each interface, though not in identical manners. To begin with, neither driver uses any firmware or USC interrupts for its own purposes. The only exception to this is in the monolithic driver's support for the HDLC serial protocol. When the HDLC Protocol Library is used, various USC interrupts are used to ensure reliable frame transmission and reception. Otherwise, all SIO4 interrupts are available for application use. The GSCAPI provides explicit support only for firmware interrupts, though it does distinguish between those generated by each channel. The monolithic driver provides explicit support for all firmware and all USC interrupts, and it does so separately for each channel.

2.36. Multithreaded Access

Both drivers support multithreaded access to their interfaces. With the GSCAPI, simultaneous access is essentially unrestricted. Any number of threads can gain simultaneous access to any board resource, except for access to the DMA engines. The driver prevents simultaneous access to the DMA engines. With the monolithic driver access is serialized by category, though all channels are serialized separately. With this mechanism each request gets exclusive device access until the service returns. If a subsequent, simultaneous call is made then it is blocked until the first call exits. Read calls are collectively serialized. Write calls are collectively serialized. IOCTL calls are collectively serialized with the exception of Wait Event requests. There is no restriction on the number of simultaneously active Wait Event requests. Wait Event requests are collectively serialized with the IOCTL calls, but only while each request is being queued or dequeued.

2.37. Register Access

Access to device registers differs between the drivers. The GSCAPI provides read and write access to both the firmware registers and to the USC registers. There's one function for reading firmware registers and another for reading USC registers, plus a function for writing to firmware registers and yet another for writing to USC registers. Macros are provided for the firmware registers, but applications are responsible for performing the arithmetic needed to access the register for the desired channel. Macros are provided for the USC registers, but no calculations are required as the functions include an argument which identifies the channel to be accessed. The register access functions are documented as being for diagnostic purposes only. This interface does not provide support for accessing PCI or PLX registers. The monolithic driver provides a more streamlined interface. First, macros are provided for all registers. This includes all firmware registers, all USC registers, all PCI registers and all PLX registers. The interface contains one IOCTL service for reading registers, one for writing to registers and one for performing read-modify-write operations. (At the application level all PCI and PLX registers are read-only.) No arithmetic is needed to access registers for the channel being accessed as the register macro encodings include all information needed by the driver to access the correct channel. In addition, firmware registers which contain content for multiple channels are processed such that the application sees its channel data right justified to the lowest significant bits. There is also a raw-register read service for reading firmware register locations by specifying only the location offset.

2.38. Sample Applications

Sample applications are included with each driver release. There is little if any similarity between the sample sets. All sample applications are designed for text-based output. The GSCAPI samples tend to make direct calls to the API. The monolithic driver samples tend to call intermediate utility services designed for generation of structured screen output. All such services are named so as to indicate the interface feature being accessed.

3. Source Code Porting

For all references to monolithic driver services refer to the driver reference manual for clarification.

3.1. Macros

3.1.1. Firmware Register Definitions

The following presents porting information for the firmware register macro definitions.

> **NOTE:** The GSCAPI defines register macros for the CTC custom firmware protocol registers. These are not supported by the monolithic driver.

GSCAPI	Monolithic	Register
`BOARD_CONTROL_REG`	`SIO4_GSC_BCR`	Board Control Register
`BOARD_STATUS_REG`	`SIO4_GSC_BSR`	Board Status Register
`CLOCK_CONTROL_REG`	`SIO4_GSC_CCR`	Clock Control Register
`CONTROL_STATUS_BASE_REG`	`SIO4_GSC_CSR`	Control/Status Register
`DATA_FIFO_BASE_REG`	`SIO4_GSC_FDR`	FIFO Data Register
`FEATURES_REG`	`SIO4_GSC_FR`	Features Register
`FIFO_COUNT_BASE_REG`	`SIO4_GSC_FCR`	FIFO Count Register
`FIFO_SIZE_BASE_REG`	`SIO4_GSC_FSR`	FIFO Size Register
`FW_REVISION_REG`	`SIO4_GSC_FRR`	Firmware Revision Register
`FW_TYPE_REG`	`SIO4_GSC_FTR`	Firmware Type Register
`INTERRUPT_CONTROL_REG`	`SIO4_GSC_ICR`	Interrupt Control Register
`INTERRUPT_EDGE_LEVEL_REG`	`SIO4_GSC_IELR`	Interrupt Edge/Level Register
`INTERRUPT_HI_LO_REG`	`SIO4_GSC_IHLR`	Interrupt High/Low Register
`INTERRUPT_STATUS_REG`	`SIO4_GSC_ISR`	Interrupt Status Register
`MAX_RX_COUNT_BASE_REG`	None Defined	Max Rx Count Register (-BiSync16 only)
`PIN_SOURCE_BASE_REG`	`SIO4_GSC_PSRCR`	Pin Source Register
`PIN_STATUS_BASE_REG`	`SIO4_GSC_PSTSR`	Pin Status Register
`POSC_CONTROL_STATUS_REG`	`SIO4_GSC_POCSR`	Programmable Osc Control/Status Register
`POSC_RAM_ADDRESS_REG`	`SIO4_GSC_PORAR`	Programmable Osc RAM Address Register
`POSC_RAM_DATA_REG`	`SIO4_GSC_PORDR`	Programmable Osc RAM Data Register
`POSC_RAM2_DATA_REG`	`SIO4_GSC_PORD2R`	Programmable Osc RAM Data 2 Register
`RX_ALMOST_BASE_REG`	`SIO4_GSC_RAR`	Rx Almost Register
`RX_COUNT_BASE_REG`	`SIO4_GSC_RCR`	Rx Count Register
`RX_PACKETINFO_FIFO_BASE_REG`	None Defined	Rx Packet Info FIFO On SIO4BXR
`SYNC_CHARACTER_BASE_REG`	`SIO4_GSC_SBR`	Sync Byte Register
`TX_ALMOST_BASE_REG`	`SIO4_GSC_TAR`	Tx Almost Register
`TX_COUNT_BASE_REG`	`SIO4_GSC_TCR`	Tx Count Register

3.1.2. USC Register Definitions

The following presents porting information for the USC register macro definitions.

GSCAPI	Monolithic	Register
`USC_CCAR`	`SIO4_USC_CCAR`	Channel Command/Address Register
`USC_CCR`	`SIO4_USC_CCR`	Channel Control Register
`USC_CCSR`	`SIO4_USC_CCSR`	Channel Command/Status Register
`USC_CMCR`	`SIO4_USC_CMCR`	Clock Mode Control Register
`USC_CMR`	`SIO4_USC_CMR`	Channel Mode Register
`USC_DCCR`	`SIO4_USC_DCCR`	Daisy Chain Control Register

`USC_ICR`	`SIO4_USC_ICR`	Interrupt Control Register
`USC_IOCR`	`SIO4_USC_IOCR`	I/O Control Register
`USC_IVR`	`SIO4_USC_IVR`	Interrupt Vector Register
`USC_HCR`	`SIO4_USC_HCR`	Hardware Configuration Register
`USC_MISR`	`SIO4_USC_MISR`	Misc Interrupt Status Register
`USC_RCCR`	`SIO4_USC_RCCR`	Receive Character Count Register
`USC_RCLR`	`SIO4_USC_RCLR`	Receive Count Limit Register
`USC_RCSR`	`SIO4_USC_RCSR`	Receive Command Status Register
`USC_RDR`	`SIO4_USC_RDR`	Receive Data Register (RO)
`USC_RICR`	`SIO4_USC_RICR`	Receive Interrupt Control Register
`USC_RMR`	`SIO4_USC_RMR`	Receive Mode Register
`USC_RSR`	`SIO4_USC_RSR`	Receive Sync Register
`USC_SICR`	`SIO4_USC_SICR`	Status Interrupt Control Register
`USC_TC0R`	`SIO4_USC_TC0R`	Time Constant 0 Register
`USC_TC1R`	`SIO4_USC_TC1R`	Time Constant 1 Register
`USC_TCCR`	`SIO4_USC_TCCR`	Transmit Character Count Register
`USC_TCLR`	`SIO4_USC_TCLR`	Transmit Count Limit Register
`USC_TCSR`	`SIO4_USC_TCSR`	Transmit Command Status Register
`USC_TDR`	`SIO4_USC_TDR`	Transmit Data Register (WO)
`USC_TICR`	`SIO4_USC_TICR`	Transmit Interrupt Control Register
`USC_TMCR`	`SIO4_USC_TMCR`	Test Mode Control Register
`USC_TMDR`	`SIO4_USC_TMDR`	Test Mode Data Register
`USC_TMR`	`SIO4_USC_TMR`	Transmit Mode Register
`USC_TSR`	`SIO4_USC_TSR`	Transmit Sync Register

3.2. System Level Routines

3.2.1. API Initialization: sio4_init()

This function is the entry point to initializing the monolithic driver’s API Library and must be the first call into the Library. This function may be called more than once, but only the first successful call actually initializes the library. Subsequent calls perform no operation at all. All other monolithic driver interface calls return a failure status when the API Library is uninitialized. Refer to the API Reference Manual for clarification.

Prototype

```
int sio4_init(void);
```

Return Value	**Description**
`0`	The operation succeeded.
`< 0`	An error occurred. See error value descriptions above (section 2.24, page 14).

3.2.2. IOCTL Service: sio4_ioctl()

The monolithic driver implements access to most features by way of IOCTL services. These services are exercised via this function. In some respects this monolithic driver functionality is similar to the GSCAPI function pair GscSio4SetOption() (section 3.4.9, page 28) and GscSio4GetOption() (section 3.4.10, page 41).

> **NOTE:** Most of the SIO4 IOCTL services update settings via non-negative values. In most of these cases, passing in a value of `-1` can be used to retrieve the current setting.

Prototype

```
int sio4_ioctl(int fd, int request, void* arg);
```

Argument	Description
`fd`	This is the file descriptor of the device to access.
`request`	This specifies the desired IOCTL service to exercise.
`arg`	This is argument specific to the requested IOCTL service.

Return Value	Description
`0`	The operation succeeded.
`< 0`	An error occurred. See error value descriptions above (section 2.24, page 14).

3.2.3. GscFindBoards

GscFindBoards(…) is used to report the number of GSC SIO4 boards in the system as well as some board specific information. An application may call this function at any time. The monolithic driver provides some of the corresponding information as outlined below.

Board Count

The monolithic driver provides the count of installed SIO4 boards via the below utility service.

Prototype

```
int sio4_count_boards(int verbose, int* get);
```

Argument	Description
`verbose`	If zero, then no console output will be generated. If non-zero, then structure console output will be generated indicating the number of boards found.
`get`	This function will report the number of boards found if this argument is non-NULL.

Return Value	Description
`0`	The operation succeeded. In this case, the value reported via the `qty` argument will be valid.
`> 0`	The number of errors encountered. In this case, the value reported via the `qty` argument is zero.

GSC_DEVICES_STRUCT->busNumber

The monolithic driver does not provide this information. While bus number information can be obtained from various Linux utilities, the only way to associate the bus number with the device being accessed is by examination of the BAR registers.

GSC_DEVICES_STRUCT->slotNumber

The monolithic driver does not provide this information. While slot number information can be obtained from various Linux utilities, the only way to associate the slot number with the device being accessed is by examination of the BAR registers.

GSC_DEVICES_STRUCT->vendorId

The Vendor Id can be obtained by reading the PCI Vendor Id Register. For the SIO4 this would be either the `GSC_PCI_9080_VIDR` or the `GSC_PCI_9056_VIDR` register, depending on the model SIO4 being accessed.

These are defined in header files `gsc_pci9080.h` and `gsc_pci9056.h`, respectively, and are included automatically by including the main interface header file. (Both register macros will obtain the correct information regardless of which model SIO4 is being accessed.)

GSC_DEVICES_STRUCT->deviceId

The Device Id can be obtained by reading the PCI Device Id Register. For the SIO4 this would be either the `GSC_PCI_9080_DIDR` or the `GSC_PCI_9056_DIDR` register, depending on the model SIO4 being accessed. These are defined in header files `gsc_pci9080.h` and `gsc_pci9056.h`, respectively, and are included automatically by including the main interface header file. (Both register macros will obtain the correct information regardless of which model SIO4 is being accessed.)

3.2.4. GscGetErrorString

GscGetErrorString(…) is used to translate the error codes that are returned by the various API functions into meaningful null-terminated strings. The monolithic driver does not have a corresponding API service. However, there is a system utility that performs a corresponding functionality. This system utility is defined in the system header `string.h`.

> **NOTE:** The monolithic driver's API Library services returns negative `errno.h` values. The utility services instead return the number of errors encountered.

Prototype

```
char* strerror(int errnum);
```

Argument	Description
`errnum`	This is the error whose string description is being requested.

Return Value	Description
`!NULL`	A pointer to a string briefly describing the error.

3.3. Board Level Routines

3.3.1. GscOpen

GscOpen(…) is used to "open" the SIO4 board for operation. It should be called before any other Board or Channel Level routines and should only be called once. In the process of opening a board, all four channels are reset and the clock outputs are disabled. The corresponding monolithic driver services gains access to only a single channel at a time. This service initializes all aspects of the channel except for programming the oscillator to its default frequency. Refer to the API Library reference manual for clarification.

Prototype

```
int sio4_open(int device, int share, int* fd);
```

<table>
<tr><th>Argument</th><th>Description</th></tr>
<tr><td><code>device</code></td><td>This is the index number of the SIO4 serial channel to access. *</td></tr>
<tr><td><code>share</code></td><td>Open the device in Shared Access Mode? If non-zero the device is opened in Shared Access Mode. If zero the device is opened in Exclusive Access Mode.</td></tr>
<tr><td><code>fd</code></td><td>The device handle is returned here. The pointer cannot be NULL. Values returned are as follows.
<table>
<tr><th>Value</th><th>Description</th></tr>
</table>
</td></tr>
</table>

	`>= 0`	This is the handle to use to access the device in subsequent calls.
	`-1`	There was an error. The device is not accessible.

* Each SIO4 board contains four independent serial channels. Each board is thus accessed by four sequential channel index numbers. That is, 0 to 3 for the first board, 4 to 7 for the second, and so on.

Return Value	Description
`0`	The operation succeeded.
`< 0`	An error occurred. See error value descriptions above (section 2.24, page 14).

3.3.2. GscClose

GscClose(…) is used to “close” the SIO4 board. It should be the last API function called before the application terminates. This function releases the resources that are used by the API and driver. The corresponding monolithic driver services releases access to only a single channel at a time. This service initializes all aspects of the channel except for programming the oscillator to its default frequency. Refer to the API Library reference manual for clarification.

Prototype

```
int sio4_close(int fd);
```

Argument	Description
`fd`	This is the file descriptor of the device to be closed.

Return Value	Description
`0`	The operation succeeded.
`< 0`	An error occurred. See error value descriptions above (section 2.24, page 14).

3.3.3. GscGetInfo

GscGetInfo(…) returns general information about an SIO4 board. The information is returned in a board info structure. The monolithic driver does not have a corresponding interface function. The same information is available, but by other means. The information from the BOARD_INFO structure can be obtained as follows.

Field	Description
`apiVersion`	The monolithic driver does not maintain a version number for the interface.
`driverVersion`	The driver version number can be read from the text file `/proc/sio4`. Refer to the monolithic driver user manual for clarification.
`fpgaVersion`	This version number is available by reading and decoding the Firmware Revision Register.
`boardType`	The GSCAPI is not precise on what information this field refers to. However, the board’s full model number can be derived by the monolithic driver’s Query service (`SIO4_IOCTL_QUERY`) and several of its options (`SIO4_QUERY_XXX`).

3.3.4. GscGetVersions

GscGetVersions(…) returns the various version numbers associated with the API, the low-level driver, and the SIO4 board’s FPGA. The monolithic driver does not have a corresponding interface function. The same information is available, but by other means. The information from the arguments can be obtained as follows.

Field	Description
`libVersion`	The monolithic driver does not maintain a version number for the libraries.
`driverVersion`	The driver version number can be read from the text file `/proc/sio4`. Refer to the monolithic driver user man for clarification.

`fpgaVersion`	This version number is available by reading and decoding the Firmware Revision Register.

3.3.5. GscLocalRegisterRead

GscLocalRegisterRead(…) is used to read the local board registers. These registers reside within the board's FPGA. The corresponding monolithic driver functionality is accessed using the IOCTL service described below. This service reads firmware registers, USC registers, PCI registers and PLX registers.

> **NOTE:** With the monolithic driver, registers which contain information for multiple channels are processed such that the data for the channel being accessed is right justifies against the lowest significant register bit.

sio4_ioctl() Usage

Argument	**Description**
`request`	`SIO4_IOCTL_REG_READ`
`arg`	`gsc_reg_t*`

Definition

```
typedef struct
{
    u32 reg;
    u32 value;
    u32 mask;
} gsc_reg_t;
```

Fields	**Description**
`reg`	This is set to the macro for the register to access.
`value`	This is the value read from the specified register.
`mask`	This is ignored for read request.

3.3.6. GscLocalRegisterWrite

GscLocalRegisterWrite(…) is used to write to the local board registers. These registers reside within the board's FPGA. The corresponding monolithic driver functionality is accessed using the IOCTL service described below. This service writes to firmware and USC registers. The PLX and PCI registers are read-only.

> **NOTE:** With the monolithic driver, registers which contain information for multiple channels are processed such that the data for the channel being accessed is right justifies against the lowest significant register bit.

sio4_ioctl() Usage

Argument	**Description**
`request`	`SIO4_IOCTL_REG_WRITE`
`arg`	`gsc_reg_t*`

Definition

```
typedef struct
{
    u32 reg;
    u32 value;
    u32 mask;
} gsc_reg_t;
```

Fields	Description
`reg`	This is set to the identifier for the register to access.
`value`	This is the value to write to the specified register.
`mask`	This is ignored for write request.

3.3.7. GscAllocPhysicalMemory

GscAllocPhysicalMemory(…) is used to attempt to allocate a physically contiguous, page-locked buffer which is safe for use with DMA operations. The monolithic driver does not have any corresponding functionality. I/O is performed via internal driver buffers. Refer to section 2.31, page 16.

3.3.8. GscMapPhysicalMemory

GscMapPhysicalMemory(…) is used to map into user virtual space a buffer previously allocated with GscAllocPhysicalMemory. The monolithic driver does not have any corresponding functionality. I/O is performed via internal driver buffers. Refer to section 2.31, page 16.

3.3.9. GscUnmapPhysicalMemory

GscUnmapPhysicalMemory(…) is used to unmap a buffer previously mapped into user virtual space with GscMapPhysicalMemory. The monolithic driver does not have any corresponding functionality. I/O is performed via internal driver buffers. Refer to section 2.31, page 16.

3.4. Channel Level Routines

3.4.1. GscSio4ChannelReset

GscSio4ChannelReset(…) resets a single channel on the SIO4 board. In addition to disabling the serial channel, this function sets the “Almost Empty” and “Almost Full” FIFO flags to 16. The corresponding monolithic driver functionality is accessed using the IOCTL service described below. This service resets all hardware and software settings to their defaults. The Almost Full and Almost Empty levels are set to their hardware defaults. The service does not adjust the programmable oscillator.

sio4_ioctl() Usage

Argument	Description
`request`	`SIO4_IOCTL_INITIALIZE`
`arg`	`s32*`

The table below lists the options used with this service.

Values	Passed to Driver	Returned by Driver
`-1`	Requests support status.	The service is unsupported.
`1`	Perform initialization.	The service is supported or initialization was performed.

3.4.2. GscSio4ChannelResetRxFifo

GscSio4ChannelResetRxFifo(…) resets the Rx FIFO for a single channel. After the reset, the FIFO will contain no data. The corresponding monolithic driver functionality is accessed using the IOCTL service described below.

sio4_ioctl() Usage

Argument	Description
`request`	`SIO4_IOCTL_RX_FIFO_RESET`
`arg`	`s32*`

The table below lists the options used with this service.

Values	Passed to Driver	Returned by Driver
`-1`	Requests support status.	The service is unsupported.
`1`	Reset the FIFO.	The service is supported or the FIFO was reset.

3.4.3. GscSio4ChannelResetTxFifo

GscSio4ChannelResetTxFifo(…) resets the Tx FIFO for a single channel. After the reset, the FIFO will contain no data. The corresponding monolithic driver functionality is accessed using the IOCTL service described below.

sio4_ioctl() Usage

Argument	Description
`request`	`SIO4_IOCTL_TX_FIFO_RESET`
`arg`	`s32*`

The table below lists the options used with this service.

Values	Passed to Driver	Returned by Driver
`-1`	Requests support status.	The service is unsupported.
`1`	Reset the FIFO.	The service is supported or the FIFO was reset.

3.4.4. GscSio4ChannelRegisterRead

GscSio4ChannelRegisterRead(…) is used to read the registers in the Universal Serial Chip that controls the specified channel. It is not recommended that a user application directly access these registers. This function is included for diagnostic purposes only. The corresponding monolithic driver functionality is accessed using the IOCTL service described below. This service reads firmware registers, USC registers, PCI registers and PLX registers.

> **NOTE:** With the monolithic driver, registers which contain information for multiple channels are processed such that the data for the channel being accessed is right justifies against the lowest significant register bit.

sio4_ioctl() Usage

Argument	Description
`request`	`SIO4_IOCTL_REG_READ`
`arg`	`gsc_reg_t*`

Definition

```
typedef struct
{
    u32 reg;
    u32 value;
    u32 mask;
} gsc_reg_t;
```

Fields	Description
`reg`	This is set to the macro for the register to access.
`value`	This is the value read from the specified register.
`mask`	This is ignored for read request.

3.4.5. GscSio4ChannelRegisterWrite

GscSio4ChannelRegisterWrite(…) is used to write to the registers in the Universal Serial Chip that controls the specified channel. The corresponding monolithic driver functionality is accessed using the IOCTL service described below. This service writes to firmware and USC registers. The PLX and PCI registers are read-only.

> **NOTE:** With the monolithic driver, registers which contain information for multiple channels are processed such that the data for the channel being accessed is right justifies against the lowest significant register bit.

sio4_ioctl() Usage

Argument	Description
`request`	`SIO4_IOCTL_REG_WRITE`
`arg`	`gsc_reg_t*`

Definition

```
typedef struct
{
    u32 reg;
    u32 value;
    u32 mask;
} gsc_reg_t;
```

Fields	Description
`reg`	This is set to the identifier for the register to access.
`value`	This is the value to write to the specified register.
`mask`	This is ignored for write request.

3.4.6. GscSio4GetLastError

GscSio4GetLastError(…) is used to retrieve the error description text of the last channel-level API call made for the specified channel. The monolithic driver does not have a corresponding API service. However, there is a system utility that performs a corresponding functionality. This system utility is defined in the system header `string.h`.

> **NOTE:** The monolithic driver's API Library services returns negative `errno.h` values. The utility services instead return the number of errors encountered.

Prototype

```
char* strerror(int errnum);
```

Argument	Description
`errnum`	This is the error whose string description is being requested.

Return Value	Description
`!NULL`	A pointer to a string briefly describing the error.

3.4.7. GscSio4ChannelSetMode

GscSio4ChannelSetMode(…) sets a single channel of the SIO4 board to the desired serial format and bitrate. See below for the corresponding functionality.

mode

On SIO4s whose firmware support it, the GSCAPI will select between the Zilog and -SYNC firmware as needed to implement the caller's request. This functionality is not supported by the monolithic driver. The corresponding monolithic driver functionality that is supported is accessed using the IOCTL services described below independently for the receiver and transmitter, respectively. This is limited to selecting between the serial protocols supported by the Zilog USC.

sio4_ioctl() Usage

Argument	Option	Description
`request`	`SIO4_IOCTL_USC_RX_MODE`	Selects the receive mode.
	`SIO4_IOCTL_USC_TX_MODE`	Selects the transmit mode.
`arg`	`s32*`	

Value Options

The GSCAPI supports the options given in the below table. The monolithic driver supports the below options independently for both the transmitter and receiver configuration.

GSCAPI	Monolithic	Description
	`-1`	Requests current setting.
`GSC_MODE_802_3`	`SIO4_USC_MODE_8023`	The 802.3 protocol.
`GSC_MODE_ASYNC`	`SIO4_USC_MODE_ASYNC`	The Asynchronous protocol.
`GSC_MODE_ASYNC_CV`	`SIO4_USC_MODE_ASY_CV`	The Asynchronous protocol with Code Violations.
`GSC_MODE_BISYNC`	`SIO4_USC_MODE_BSC`	The Bisync protocol.
`GSC_MODE_HDLC`	`SIO4_USC_MODE_HDLC`	The HDLC protocol.
`GSC_MODE_ISO`	`SIO4_USC_MODE_ISOC`	The Isochronous protocol.
`GSC_MODE_MONOSYNC`	`SIO4_USC_MODE_MONO`	The Monosync protocol.
`GSC_MODE_NBIF`	`SIO4_USC_MODE_NBIP`	The Nine-Bit Interprocessor Protocol.
`GSC_MODE_SYNC`	`SIO4_USC_MODE_E_SYNC`	The External Synchronous protocol.
`GSC_MODE_SYNC_ENV`	Not selectable. The default mode when using a -SYNC model board.	The -SYNC model protocol.
`GSC_MODE_TRANS_BISYNC`	`SIO4_USC_MODE_TBSC`	The Transparent Bisync protocol.

bitRate

The configuration necessary for any given bitrate depends on the serial protocol, the desired bitrate and the cable signals in use. The current protocol also dictates the upper and lower bitrate limits that the SIO4 can generate. The functionality required to derive the configuration needed for a given setup is not available via a single service and is protocol dependent.

Asynchronous Protocol

The configuration and bitrate limits for the Asynchronous protocol are embedded inside the Asynchronous Protocol Library. Refer to the Asynchronous Protocol Library Reference Manual for clarification.

HDLC Protocol

The configuration and bitrate limits for the HDLC protocol are embedded inside the HDLC Protocol Library. Refer to the HDLC Protocol Library Reference Manual for clarification.

Isochronous Protocol

The configuration and bitrate limits for the Isochronous protocol are embedded inside the Isochronous Protocol Library. Refer to the Isochronous Protocol Library Reference Manual for clarification.

SYNC Protocol

The configuration and bitrate limits for the SYNC protocol are embedded inside the SYNC Protocol Library. Refer to the SYNC Protocol Library Reference Manual for clarification.

Other Protocols

For information on the limits and means of configuring the SIO4 for other protocols, please refer to the SIO4 board user manual and the Zilog User Manual.

3.4.8. GscSio4ChannelGetMode

GscSio4ChannelGetMode(…) requests the current serial channel protocol selection and bitrate. The corresponding monolithic driver functionality is not available via a single service. First, the mode can be set differently for the Tx and Rx streams. Second, computing the configuration necessary for the desired bitrate can involve floating point operations, which are not permitted inside Linux drivers.

mode

Refer to the Channel Mode portion of the GscSio4ChannelSetMode section above (section 3.4.7, page 27). Passing in an argument value of `-1` will retrieve the current setting.

bitRate

The monolithic driver neither records the bitrate nor computes the rate from a channel's current configuration. Protocol specific services are provided that compute the necessary configuration for a desired bitrate, but not the reverse.

3.4.9. GscSio4SetOption

GscSio4SetOption(…) sets the value of a protocol configuration option for a channel. The available options are defined by the GSC_OPTION_NAME enumerated type. The corresponding monolithic driver functionality is accessed using the `sio4_ioctl()` service (section 3.2.2, page 19). The set of `GSC_OPTION_NAME` options from the GSCAPI are represented by corresponding IOCTL services from the monolithic driver. Also, each driver has a corresponding set of option values. The options and their values are presented in the tables below.

3.4.9.1. GSC_SIO_DATASIZE

This service configures the word size. The GSCAPI has one value range for Zilog devices and another for SYNC boards. The monolithic driver uses IOCTL services for the same purpose. Also, the GSCAPI encodes the Tx and Rx options differently for the set and get services, while the monolithic driver uses the `s32*` type for all options.

	GSCAPI	Monolithic	Description
Option	`GSC_SIO_DATASIZE`	`SIO4_IOCTL_USC_RX_CHAR_LEN` `SIO4_IOCTL_USC_TX_CHAR_LEN`	For Zilog firmware models.

Values		`-1`	Retrieve the current setting.

	`1 – 8`	`SIO4_USC_CHAR_LEN_1`
		…
		`SIO4_USC_CHAR_LEN_8`
			Option to apply.

	GSCAPI	Monolithic	Description
Option	`GSC_SIO_DATASIZE`	`SIO4_IOCTL_SYNC_TX_WORD_SIZE`	For -SYNC firmware models.

Values		`-1`	Retrieve the current setting.
	`0 – 65,535`	`0 – 65,535`	Option to apply.

3.4.9.2. GSC_SIO_GAPSIZE

The size of the gap between transmitted data words for a single channel of the SIO4 board. The monolithic driver uses an IOCTL service for the same purpose.

	GSCAPI	Monolithic	Description
Option	`GSC_SIO_GAPSIZE`	`SIO4_IOCTL_SYNC_TX_GAP_SIZE`	For -SYNC firmware models.

Values		`-1`	Retrieve the current setting.
	`0 – 65,535`	`0 – 65,535`	Option to apply.

3.4.9.3. GSC_SIO_MSBLSBORDER

The byte ordering of both transmitted and received data words for a single channel of the SIO4 board. The monolithic driver uses IOCTL services for the same purpose. Also, the GSCAPI encodes the Tx and Rx options differently for the set and get services, while the monolithic driver uses the `s32*` type for all options.

	GSCAPI	Monolithic	Description
Option	`GSC_SIO_MSBLSBORDER`	`SIO4_IOCTL_USC_SEND_COMMAND`	For Zilog firmware models.

Values		`-1`	Retrieve the current setting.
	`GSC_MSB_FIRST`	`SIO4_USC_SEND_CMD_SEL_MSB_FIRST`	Option to apply.
	`GSC_LSB_FIRST`	`SIO4_USC_SEND_CMD_SEL_LSB_FIRST`	Option to apply.

	GSCAPI	Monolithic	Description
Option	`GSC_SIO_MSBLSBORDER`	`SIO4_IOCTL_SYNC_RX_BIT_ORDER` `SIO4_IOCTL_SYNC_TX_BIT_ORDER`	For -SYNC firmware models.

Values		`-1`	Retrieve the current setting.
	`GSC_MSB_FIRST`	`SIO4_SYNC_BIT_ORDER_MSB`	Option to apply.
	`GSC_LSB_FIRST`	`SIO4_SYNC_BIT_ORDER_LSB`	Option to apply.

3.4.9.4. GSC_SIO_PARITY

The type of parity that will be used on a single channel of the SIO4 board. The monolithic driver uses IOCTL services for the same purpose. In addition, the monolithic driver uses one service pair to enable and disable the use of parity while another service pair is used to select the parity type.

	GSCAPI	Monolithic	Description
Option	`GSC_SIO_PARITY`	`SIO4_IOCTL_USC_RX_PAR_ENABLE` `SIO4_IOCTL_USC_TX_PAR_ENABLE`	For Zilog firmware models.

Values		`-1`	Retrieve the current setting.
	`GSC_PARITY_NONE`	`SIO4_USC_PAR_ENABLE_NO`	Option to apply.
		`SIO4_USC_PAR_ENABLE_YES`	Option to apply.

	GSCAPI	Monolithic	Description
Option	`GSC_SIO_PARITY`	`SIO4_IOCTL_USC_RX_PAR_TYPE` `SIO4_IOCTL_USC_TX_PAR_TYPE`	For Zilog firmware models.

Values		`-1`	Retrieve the current setting.
	`GSC_PARITY_EVEN`	`SIO4_USC_PAR_TYPE_EVEN`	Option to apply.
	`GSC_PARITY_ODD`	`SIO4_USC_PAR_TYPE_ODD`	Option to apply.
	`GSC_PARITY_MARK`	`SIO4_USC_PAR_TYPE_ONE`	Option to apply.
	`GSC_PARITY_SPACE`	`SIO4_USC_PAR_TYPE_ZERO`	Option to apply.

3.4.9.5. GSC_SIO_STOPBITS

The number of stop bits to use for a single channel of the SIO4 board. The monolithic driver uses an IOCTL service for the same purpose. The minimum valid stop bit size is 9/16th of a bit wide. In addition, the Zilog chip support adjustable stop bit widths from 9/16th of a bit to two-bits wide in 1/6th increments.

	GSCAPI	Monolithic	Description
Option	`GSC_SIO_STOPBITS`	`SIO4_IOCTL_USC_ASYNC_TX_STOP_BIT`	For Zilog firmware models.

Values		`-1`	Retrieve the current setting.
	`GSC_STOP_BITS_0`	Not an acceptable option.	Option to apply.

		SIO4_USC_ASYNC_TX_STOP_BIT_0__9_16 … SIO4_USC_ASYNC_TX_STOP_BIT_0_15_16	Option to apply.
	GSC_STOP_BITS_1	SIO4_USC_ASYNC_TX_STOP_BIT_1	Option to apply.
		SIO4_USC_ASYNC_TX_STOP_BIT_1__1_16 … SIO4_USC_ASYNC_TX_STOP_BIT_1__7_16	Option to apply.
	GSC_STOP_BITS_1_5	SIO4_USC_ASYNC_TX_STOP_BIT_1__8_16	Option to apply.
		SIO4_USC_ASYNC_TX_STOP_BIT_1__9_16 … SIO4_USC_ASYNC_TX_STOP_BIT_1_15_16	Option to apply.
	GSC_STOP_BITS_2	SIO4_USC_ASYNC_TX_STOP_BIT_2	Option to apply.

3.4.9.6. GSC_SIO_ENCODING

The encoding type for a single channel of the SIO4 board. The monolithic driver uses IOCTL services for the same purpose.

	GSCAPI	Monolithic	Description
Option	GSC_SIO_ENCODING	SIO4_IOCTL_USC_RX_DATA_ENCODE SIO4_IOCTL_USC_TX_DATA_ENCODE	For Zilog firmware models.

Values		-1	Retrieve the current setting.
	GSC_ENCODING_NRZ	SIO4_USC_DATA_ENCODE_NRZ	Option to apply.
	GSC_ENCODING_NRZB	SIO4_USC_DATA_ENCODE_NRZB	Option to apply.
	GSC_ENCODING_NRZI_MARK	SIO4_USC_DATA_ENCODE_NRZI_MARK	Option to apply.
	GSC_ENCODING_NRZI_SPACE	SIO4_USC_DATA_ENCODE_NRZI_SPACE	Option to apply.
	GSC_ENCODING_BIPHASE_MARK	SIO4_USC_DATA_ENCODE_BI_MARK	Option to apply.
	GSC_ENCODING_BIPHASE_SPACE	SIO4_USC_DATA_ENCODE_BI_SPACE	Option to apply.
	GSC_ENCODING_BIPHASE_LEVEL	SIO4_USC_DATA_ENCODE_BI_LEVEL	Option to apply.
	GSC_ENCODING_DIFF_BIPHASE_LEVEL	SIO4_USC_DATA_ENCODE_D_BI_LEVEL	Option to apply.

3.4.9.7. GSC_SIO_PROTOCOL

The physical interface protocol and termination options. The monolithic driver uses IOCTL services for the same purpose. Also, the GSCAPI encodes the protocol and termination options differently for the set and get services, while the monolithic driver uses the `s32*` type for all options. In addition, the monolithic driver uses one service pair for the protocol and another pair for the termination. Plus, as the monolithic driver supports more SIO4 models, there are other options that may be reported by the driver.

	GSCAPI	Monolithic	Description
Option	GSC_SIO_PROTOCOL	SIO4_IOCTL_XCVR_PROTOCOL	

Values		-1	Retrieve the current setting.
	GSC_PROTOCOL_RS422_RS485	SIO4_XCVR_PROTOCOL_RS422_RS485	Option to apply.
	GSC_PROTOCOL_RS423	SIO4_XCVR_PROTOCOL_RS423	Option to apply.

	`GSC_PROTOCOL_RS232`	`SIO4_XCVR_PROTOCOL_RS232`	Option to apply.
	`GSC_PROTOCOL_RS530_1`	`SIO4_XCVR_PROTOCOL_RS530`	Option to apply.
	`GSC_PROTOCOL_RS530_2`	`SIO4_XCVR_PROTOCOL_RS530A`	Option to apply.
	`GSC_PROTOCOL_V35_1`	`SIO4_XCVR_PROTOCOL_V35`	Option to apply.
	`GSC_PROTOCOL_V35_2`	`SIO4_XCVR_PROTOCOL_V35A`	Option to apply.
	`GSC_PROTOCOL_RS422_RS423_1`	`SIO4_XCVR_PROTOCOL_RS422_RS423_1`	Option to apply.
	`GSC_PROTOCOL_RS422_RS423_2`	`SIO4_XCVR_PROTOCOL_RS422_RS423_2`	Option to apply.
		`SIO4_XCVR_PROTOCOL_INVALID`	A reported option.
		`SIO4_XCVR_PROTOCOL_UNKNOWN`	A reported option.

	GSCAPI	Monolithic	Description
Option	`GSC_SIO_PROTOCOL`	`SIO4_IOCTL_XCVR_TERM`	

Values		`-1`	Retrieve the current setting.
	`GSC_TERMINATION_ENABLED`	`SIO4_XCVR_TERM_ENABLE`	Option to apply.
	`GSC_TERMINATION_DISABLED`	`SIO4_XCVR_TERM_DISABLE`	Option to apply.

3.4.9.8. GSC_SIO_DTEDCE

Sets a single channel of the SIO4 board to either DTE or DCE mode. The monolithic driver uses an IOCTL service for the same purpose.

	GSCAPI	Monolithic	Description
Option	`GSC_SIO_DTEDCE`	`SIO4_IOCTL_CBL_MODE`	For Zilog firmware models.

Values		`-1`	Retrieve the current setting.
	`GSC_PIN_DTE`	`SIO4_CBL_MODE_DTE`	Option to apply.
	`GSC_PIN_DCE`	`SIO4_CBL_MODE_DCE`	Option to apply.

3.4.9.9. GSC_SIO_LOOPBACK

Sets he loopback mode of a channel on the SIO4 board. The monolithic driver uses an IOCTL service for the same purpose. In addition, the monolithic driver has another selection option, when supported by the SIO4 being accessed.

	GSCAPI	Monolithic	Description
Option	`GSC_SIO_LOOPBACK`	`SIO4_IOCTL_LOOP_BACK`	For Zilog firmware models.

Values		`-1`	Retrieve the current setting.
	`GSC_LOOP_NONE`	`SIO4_LOOP_BACK_DISABLE`	Option to apply.
	`GSC_LOOP_EXTERNAL`	`SIO4_LOOP_BACK_EXTERNAL`	Option to apply.
		`SIO4_LOOP_BACK_INTERNAL`	Option to apply.

3.4.9.10. GSC_SIO_RECEIVER

Used for enabling or disabling the receiver for a single channel on the SIO4 board. The monolithic driver uses IOCTL services for the same purpose. Plus, the monolithic driver has additional selections.

	GSCAPI	**Monolithic**	**Description**
Option	`GSC_SIO_RECEIVER`	`SIO4_IOCTL_USC_RX_ENABLE`	For Zilog firmware models.

Values		`-1`	Retrieve the current setting.
	`GSC_ENABLED`	`SIO4_USC_ENABLE_YES_NOW`	Option to apply.
		`SIO4_USC_ENABLE_YES_W_AE`	Option to apply.
	`GSC_DISABLED`	`SIO4_USC_ENABLE_NO_NOW`	Option to apply.
		`SIO4_USC_ENABLE_NO_AFTER`	Option to apply.

	GSCAPI	**Monolithic**	**Description**
Option	`GSC_SIO_RECEIVER`	`SIO4_IOCTL_SYNC_RX_ENABLE`	For -SYNC firmware models.

Values		`-1`	Retrieve the current setting.
	`GSC_ENABLED`	`SIO4_SYNC_ENABLE_YES`	Option to apply.
	`GSC_DISABLED`	`SIO4_SYNC_ENABLE_NO`	Option to apply.

3.4.9.11. GSC_SIO_TRANSMITTER

Used for enabling or disabling the receiver for a single channel on the SIO4 board. The monolithic driver uses IOCTL services for the same purpose.

	GSCAPI	**Monolithic**	**Description**
Option	`GSC_SIO_TRANSMITTER`	`SIO4_IOCTL_USC_TX_ENABLE`	For Zilog firmware models.

Values		`-1`	Retrieve the current setting.
	`GSC_ENABLED`	`SIO4_USC_ENABLE_YES_NOW`	Option to apply.
		`SIO4_USC_ENABLE_YES_W_AE`	Option to apply.
	`GSC_DISABLED`	`SIO4_USC_ENABLE_NO_NOW`	Option to apply.
		`SIO4_USC_ENABLE_NO_AFTER`	Option to apply.

	GSCAPI	**Monolithic**	**Description**
Option	`GSC_SIO_TRANSMITTER`	`SIO4_IOCTL_SYNC_TX_ENABLE`	For -SYNC firmware models.

Values		`-1`	Retrieve the current setting.
	`GSC_ENABLED`	`SIO4_SYNC_ENABLE_YES`	Option to apply.
	`GSC_DISABLED`	`SIO4_SYNC_ENABLE_NO`	Option to apply.

3.4.9.12. GSC_SIO_TXDATAPINMODE

Used to enable the TxD pin of a channel to be used for general purpose I/O. The monolithic driver uses an IOCTL service for the same purpose. In addition, the monolithic driver has additional selection options.

	GSCAPI	Monolithic	Description
Option	`GSC_SIO_TXDATAPINMODE`	`SIO4_IOCTL_Z16_CBL_TXD_CFG`	For Zilog firmware models.

Values		`-1`	Retrieve the current setting.
	`GSC_PIN_AUTO`	`SIO4_Z16_CBL_TXD_CFG_OUT_USC_TXD`	Option to apply.
	`GSC_PIN_GPIO`	`SIO4_Z16_CBL_TXD_CFG_OUT_1`	Option to apply.
		`SIO4_Z16_CBL_TXD_CFG_OUT_0`	Option to apply.

	GSCAPI	Monolithic	Description
Option	`GSC_SIO_TXDATAPINMODE`	`SIO4_IOCTL_SYNC_TXD_CFG`	For -SYNC firmware models.

Values		`-1`	Retrieve the current setting.
	`GSC_PIN_AUTO`	`SIO4_SYNC_TXD_CFG_ACTIVE_HI`	Option to apply. (NRZ)
		`SIO4_SYNC_TXD_CFG_ACTIVE_LO`	Option to apply. (NRZB)
	`GSC_PIN_GPIO`	`SIO4_SYNC_TXD_CFG_OUT_1`	Option to apply.
		`SIO4_SYNC_TXD_CFG_OUT_0`	Option to apply.

3.4.9.13. GSC_SIO_RXDATAPINMODE

Used to enable the RxD pin of a channel to be used for general purpose I/O. The RxD cable signal is not configurable the way it is indicated in the GSCAPI documentation. However, the monolithic driver provides an IOCTL service to configure the RxD signal as supported by firmware.

	GSCAPI	Monolithic	Description
Option	`GSC_SIO_RXDATAPINMODE`	No corresponding IOCTL service.	For Zilog firmware models.

Values	`GSC_PIN_AUTO`	Not a supported capability.	N/A
	`GSC_PIN_GPIO`	Not a supported capability.	N/A

	GSCAPI	Monolithic	Description
Option	`GSC_SIO_RXDATAPINMODE`	`SIO4_IOCTL_SYNC_RXD_CFG`	For -SYNC firmware models.

Values		`-1`	Retrieve the current setting.
	`GSC_PIN_AUTO`	`SIO4_SYNC_RXD_CFG_ACTIVE_HI`	Option to apply. (NRZ)
	`GSC_PIN_GPIO`	Not a supported capability.	Option to apply.
		`SIO4_SYNC_RXD_CFG_ACTIVE_LO`	Option to apply. (NRZB)

3.4.9.14. GSC_SIO_TXCLOCKPINMODE

Used to enable the TxC pin of a channel to be used for general purpose I/O. The monolithic driver uses IOCTL services for the same purpose. In addition, the monolithic driver has additional selections.

	GSCAPI	Monolithic	Description
Option	`GSC_SIO_TXCLOCKPINMODE`	`SIO4_IOCTL_Z16_CBL_TXC_CFG`	For Zilog firmware models.
Values		`-1`	Retrieve the current setting.
		`SIO4_Z16_CBL_TXC_CFG_OUT_OSC`	Option to apply.
		`SIO4_Z16_CBL_TXC_CFG_OUT_OSC_INV`	Option to apply.
	`GSC_PIN_AUTO`	`SIO4_Z16_CBL_TXC_CFG_OUT_USC_TXC`	Option to apply.
		`SIO4_Z16_CBL_TXC_CFG_OUT_USC_RXC`	Option to apply.
		`SIO4_Z16_CBL_TXC_CFG_OUT_CBL_RXC`	Option to apply.
		`SIO4_Z16_CBL_TXC_CFG_OUT_CBL_RXA`	Option to apply.
		`SIO4_Z16_CBL_TXC_CFG_OUT_0`	Option to apply.
	`GSC_PIN_GPIO`	`SIO4_Z16_CBL_TXC_CFG_OUT_1`	Option to apply.

	GSCAPI	Monolithic	Description
Option	`GSC_SIO_TXCLOCKPINMODE`	`SIO4_IOCTL_SYNC_TXC_SRC`	For -SYNC firmware models.
Values		`-1`	Retrieve the current setting.
	`GSC_PIN_AUTO`	`SIO4_SYNC_TXC_SRC_OSC_HALF_RISE`	Option to apply.
		`SIO4_SYNC_TXC_SRC_OSC_HALF_FALL`	Option to apply.
		`SIO4_SYNC_TXC_SRC_EXT_RISE`	Option to apply.
		`SIO4_SYNC_TXC_SRC_EXT_FALL`	Option to apply.
		`SIO4_SYNC_TXC_SRC_0`	Option to apply.
	`GSC_PIN_GPIO`	`SIO4_SYNC_TXC_SRC_1`	Option to apply.

3.4.9.15. GSC_SIO_RXCLOCKPINMODE

Used to enable the RxC pin of a channel to be used for general purpose I/O. The monolithic driver uses an IOCTL service for the same purpose. In addition, the monolithic driver has additional selections.

	GSCAPI	Monolithic	Description
Option	`GSC_SIO_RXCLOCKPINMODE`	`SIO4_IOCTL_USC_RXC_CFG`	For Zilog firmware models.
Values		`-1`	Retrieve the current setting.
	`GSC_PIN_CLOCK_REVERSEPOLARITY` Selects an invalid firmware option.		Option to apply.
		`SIO4_USC_RXC_CFG_IN_OSC`	Option to apply.
		`SIO4_USC_RXC_CFG_IN_OSC_INV`	Option to apply.
	`GSC_PIN_AUTO`	`SIO4_USC_RXC_CFG_IN_CBL_RXC`	Option to apply.
		`SIO4_USC_RXC_CFG_IN_CBL_RXAUXC`	Option to apply.
		`SIO4_USC_RXC_CFG_OUT_RX_CLK`	Option to apply.
		`SIO4_USC_RXC_CFG_OUT_RX_BYTE_CLK`	Option to apply.
		`SIO4_USC_RXC_CFG_OUT_SYNC`	Option to apply.
		`SIO4_USC_RXC_CFG_OUT_BRG0`	Option to apply.
		`SIO4_USC_RXC_CFG_OUT_BRG1`	Option to apply.
		`SIO4_USC_RXC_CFG_OUT_CTR0`	Option to apply.
		`SIO4_USC_RXC_CFG_OUT_DPLL_RX`	Option to apply.
		`SIO4_USC_RXC_CFG_IN_0`	Option to apply.
	`GSC_PIN_GPIO`	`SIO4_USC_RXC_CFG_IN_1`	Option to apply.

3.4.9.16. GSC_SIO_CTSPINMODE

Used to enable the CTS pin of a channel to be used for general purpose I/O. The monolithic driver uses an IOCTL service for the same purpose. In addition, the monolithic driver has additional selections.

	GSCAPI	Monolithic	Description
Option	`GSC_SIO_CTSPINMODE`	`SIO4_IOCTL_USC_CTS_CFG`	For Zilog firmware models.

Values		`-1`	Retrieve the current setting.
		`SIO4_USC_CTS_CFG_TRI`	Option to apply.
	`GSC_PIN_AUTO`	`SIO4_USC_CTS_CFG_IN_CBL_CTS`	Option to apply.
		`SIO4_USC_CTS_CFG_OUT_0`	Option to apply.
		`SIO4_USC_CTS_CFG_OUT_1`	Option to apply.
	`GSC_PIN_GPIO` Not supported.		Option to apply.

3.4.9.17. GSC_SIO_RTSPINMODE

Used to enable the RTS pin of a channel to be used for general purpose I/O. The monolithic driver uses IOCTL services for the same purpose. In addition, the monolithic driver has additional selections.

	GSCAPI	Monolithic	Description
Option	`GSC_SIO_RTSPINMODE`	`SIO4_IOCTL_Z16_CBL_RTS_CFG`	For Zilog firmware models.

Values		`-1`	Retrieve the current setting.
		`SIO4_Z16_CBL_RTS_CFG_OUT_USC_CTS`	Option to apply.
	`GSC_PIN_AUTO` Selects incorrect option.	`SIO4_Z16_CBL_RTS_CFG_OUT_RTS`	Option to apply.
		`SIO4_Z16_CBL_RTS_CFG_OUT_0`	Option to apply.
	`GSC_PIN_GPIO`	`SIO4_Z16_CBL_RTS_CFG_OUT_1`	Option to apply.

	GSCAPI	Monolithic	Description
Option	`GSC_SIO_RTSPINMODE`	`SIO4_IOCTL_SYNC_TXE_CFG`	For -SYNC firmware models.

Values		`-1`	Retrieve the current setting.
	`GSC_PIN_AUTO`	`SIO4_SYNC_TXE_CFG_ACTIVE_HI`	Option to apply.
		`SIO4_SYNC_TXE_CFG_ACTIVE_LO`	Option to apply.
		`SIO4_SYNC_TXE_CFG_OUT_0`	Option to apply.
	`GSC_PIN_GPIO`	`SIO4_SYNC_TXE_CFG_OUT_1`	Option to apply.

3.4.9.18. GSC_SIO_CLOCKSOURCE

Used to set the clock pin sources of the transmitter and receiver. The monolithic driver uses IOCTL services for the same purpose. In addition, the monolithic driver has additional selections. The GSCAPI encodes the selections into a single value. The monolithic driver uses an `s32*` for all options.

	GSCAPI	Monolithic	Description
Option	`GSC_SIO_CLOCKSOURCE`	`SIO4_IOCTL_USC_TXC_CFG`	For Zilog firmware models. Tx Channel

Values		`-1`	Retrieve the current setting.
	`GSC_CLOCK_INTERNAL`	`SIO4_USC_TXC_CFG_IN_OSC`	Option to apply.
		`SIO4_USC_TXC_CFG_IN_OSC_INV`	Option to apply.
	`GSC_CLOCK_EXTERNAL`	`SIO4_USC_TXC_CFG_IN_CBL_RXC`	Option to apply.
	`GSC_CLOCK_EXT_RX_AUX`	`SIO4_USC_TXC_CFG_IN_CBL_RXAUXC`	Option to apply.
		`SIO4_USC_TXC_CFG_IN_0`	Option to apply.
		`SIO4_USC_TXC_CFG_IN_1`	Option to apply.
		`SIO4_USC_TXC_CFG_OUT_TX_CLK`	Option to apply.
		`SIO4_USC_TXC_CFG_OUT_TX_BYTE_CLK`	Option to apply.
		`SIO4_USC_TXC_CFG_OUT_TX_COMP`	Option to apply.
		`SIO4_USC_TXC_CFG_OUT_BRG0`	Option to apply.
		`SIO4_USC_TXC_CFG_OUT_BRG1`	Option to apply.
		`SIO4_USC_TXC_CFG_OUT_CTR1`	Option to apply.
		`SIO4_USC_TXC_CFG_OUT_DPLL_TX`	Option to apply.

	GSCAPI	**Monolithic**	**Description**
Option	`GSC_SIO_CLOCKSOURCE`	`SIO4_IOCTL_USC_RXC_CFG`	For Zilog firmware models. Rx Channel

Values		`-1`	Retrieve the current setting.
	`GSC_CLOCK_INTERNAL`	`SIO4_USC_RXC_CFG_IN_OSC`	Option to apply.
		`SIO4_USC_RXC_CFG_IN_OSC_INV`	Option to apply.
	`GSC_CLOCK_EXTERNAL`	`SIO4_USC_RXC_CFG_IN_CBL_RXC`	Option to apply.
	`GSC_CLOCK_EXT_RX_AUX`	`SIO4_USC_RXC_CFG_IN_CBL_RXAUXC`	Option to apply.
		`SIO4_USC_RXC_CFG_IN_0`	Option to apply.
		`SIO4_USC_RXC_CFG_IN_1`	Option to apply.
		`SIO4_USC_RXC_CFG_OUT_RX_CLK`	Option to apply.
		`SIO4_USC_RXC_CFG_OUT_RX_BYTE_CLK`	Option to apply.
		`SIO4_USC_RXC_CFG_OUT_SYNC`	Option to apply.
		`SIO4_USC_RXC_CFG_OUT_BRG0`	Option to apply.
		`SIO4_USC_RXC_CFG_OUT_BRG1`	Option to apply.
		`SIO4_USC_RXC_CFG_OUT_CTR0`	Option to apply.
		`SIO4_USC_RXC_CFG_OUT_DPLL_RX`	Option to apply.

	GSCAPI	**Monolithic**	**Description**
Option	`GSC_SIO_CLOCKSOURCE`	`SIO4_IOCTL_SYNC_TXC_SRC`	For -SYNC firmware models.

Values		`-1`	Retrieve the current setting.
	`GSC_CLOCK_INTERNAL`	`SIO4_SYNC_TXC_SRC_OSC_HALF_RISE`	Option to apply.
	`GSC_CLOCK_EXTERNAL`	`SIO4_SYNC_TXC_SRC_EXT_RISE`	Option to apply.
	`GSC_CLOCK_EXT_RX_AUX`	Option not supported by firmware.	Option to apply.
		`SIO4_SYNC_TXC_SRC_0`	Option to apply.
		`SIO4_SYNC_TXC_SRC_1`	Option to apply.

3.4.9.19. GSC_SIO_CRCMODE

The GSCAPI encodes the mode and preset value selections into a single value. The monolithic driver uses an `s32*` for all options.

CRC Mode

Used for setting the CRC generation/detection mode for a single channel. The monolithic driver uses IOCTL services for the same purpose.

	GSCAPI	Monolithic	Description
Option	`GSC_SIO_CRCMODE`	`SIO4_IOCTL_USC_RX_CRC_TYPE` `SIO4_IOCTL_USC_TX_CRC_TYPE`	For Zilog firmware models.

Values		`-1`	Retrieve the current setting.
	`GSC_CRC_16`	`SIO4_USC_CRC_TYPE_16`	Option to apply.
	`GSC_CRC_32`	`SIO4_USC_CRC_TYPE_32`	Option to apply.
	`GSC_CRC_CCITT`	`SIO4_USC_CRC_TYPE_CCITT`	Option to apply.

CRC Preset Value

Used for setting the CRC generation/detection mode for a single channel. The monolithic driver uses IOCTL services for the same purpose.

	GSCAPI	Monolithic	Description
Option	`GSC_SIO_CRCMODE`	`SIO4_IOCTL_USC_RX_CRC_PRESET` `SIO4_IOCTL_USC_TX_CRC_PRESET`	For Zilog firmware models.

Values		`-1`	Retrieve the current setting.
	`GSC_CRC_INIT_0`	`SIO4_USC_CRC_PRESET_ALL_0`	Option to apply.
	`GSC_CRC_INIT_1`	`SIO4_USC_CRC_PRESET_ALL_1`	Option to apply.

3.4.9.20. GSC_SIO_SYNCWORD

Used to set the sync word used on a channel. The monolithic driver uses IOCTL services for the same purpose. In addition, the monolithic driver has additional selections. The GSCAPI encodes the selections into a single value. The monolithic driver uses an `s32*` for all options.

	GSCAPI	Monolithic	Description
Option	`GSC_SIO_SYNCWORD`	`SIO4_IOCTL_Z16_SYNC_BYTE`	For Zilog firmware models.

Values		`-1`	Retrieve the current setting.
	`0 – 65,535`	`0 - 255`	Option to apply.

	GSCAPI	Monolithic	Description

Option	`GSC_SIO_SYNCWORD`	`SIO4_IOCTL_USC_MONO_RX_SYNC`
		`SIO4_IOCTL_USC_MONO_TX_SYNC`
		`SIO4_IOCTL_USC_SMONO_RX_SYNC`
		`SIO4_IOCTL_USC_SMONO_TX_SYNC`
			For Zilog firmware models.

Values		`-1`	Retrieve the current setting.
	`0 – 65,535`	`0 - 255`	Option to apply.

	GSCAPI	**Monolithic**	**Description**

Option	`GSC_SIO_SYNCWORD`	`SIO4_IOCTL_USC_BSC_RX_SYN0`
		`SIO4_IOCTL_USC_BSC_RX_SYN1`
		`SIO4_IOCTL_USC_BSC_TX_SYN0`
		`SIO4_IOCTL_USC_BSC_TX_SYN1`
			For Zilog firmware models.

Values		`-1`	Retrieve the current setting.
	`0 - 65,535`	`0 – 255 for lower` (SYN0) `0 - 255 for upper` (SYN1)	Option to apply.

3.4.9.21. GSC_SIO_TXUNDERRUN

Sets the data pattern to be transmitted under a Tx underrun condition. The monolithic driver uses an IOCTL service for the same purpose.

	GSCAPI	**Monolithic**	**Description**
Option	`GSC_SIO_TXUNDERRUN`	`SIO4_IOCTL_USC_BSC_TX_UNDERRUN`	For Zilog firmware models.

Values		`-1`	Retrieve the current setting.
	`GSC_SYN`	`SIO4_USC_BSC_TX_UNDERRUN_S1`	Option to apply.
	`GSC_SYN0_SYN1`	`SIO4_USC_BSC_TX_UNDERRUN_S01`	Option to apply.
	`GSC_CRC_SYN1`	`SIO4_USC_BSC_TX_UNDERRUN_CRC_S1`	Option to apply.
	`GSC_CRC_SYN0_SYN1`	`SIO4_USC_BSC_TX_UNDERRUN_CRC_S01`	Option to apply.

3.4.9.22. GSC_SIO_TXPREAMBLE

Used to enable or disable the Tx preamble for a channel. The monolithic driver uses an IOCTL service for the same purpose. The monolithic driver supports *preamble enable* services for the Rx stream and for other serial protocols.

	GSCAPI	**Monolithic**	**Description**
Option	`GSC_SIO_TXPREAMBLE`	`SIO4_IOCTL_USC_BSC_TX_PREAMBLE`	For Zilog firmware models.

Values		`-1`	Retrieve the current setting.
	`GSC_ENABLED`	`SIO4_USC_BSC_TX_PREAMBLE_YES`	Option to apply.
	`GSC_DISABLED`	`SIO4_USC_BSC_TX_PREAMBLE_NO`	Option to apply.

3.4.9.23. GSC_SIO_TXSHORTSYNC

Used to set the Tx sync length (short or 8 bit) for a channel. The monolithic driver uses an IOCTL service for the same purpose. The monolithic driver has similar *short* settings for Rx streams and other serial protocols.

	GSCAPI	**Monolithic**	**Description**
Option	`GSC_SIO_TXSHORTSYNC`	`SIO4_IOCTL_USC_BSC_TX_SHORT`	For Zilog firmware models.

Values		`-1`	Retrieve the current setting.
	`GSC_ENABLED`	`SIO4_USC_BSC_SHORT_YES`	Option to apply.
	`GSC_DISABLED`	`SIO4_USC_BSC_SHORT_NO`	Option to apply.

3.4.9.24. GSC_SIO_RXSYNCSTRIP

Set the Rx sync strip mode for a channel. The monolithic driver uses an IOCTL service for the same purpose.

	GSCAPI	Monolithic	Description
Option	GSC_SIO_RXSYNCSTRIP	SIO4_IOCTL_USC_BSC_RX_STRIP	For Zilog firmware models.

Values		-1	Retrieve the current setting.
	GSC_ENABLED	SIO4_USC_BSC_RX_STRIP_YES	Option to apply.
	GSC_DISABLED	SIO4_USC_BSC_RX_STRIP_NO	Option to apply.

3.4.9.25. GSC_SIO_RXSHORTSYNC

Used to set the Rx sync length (short or 8 bit) for a channel. The monolithic driver uses an IOCTL service for the same purpose. The monolithic driver has similar *short* settings for Tx streams and other serial protocols.

	GSCAPI	Monolithic	Description
Option	GSC_SIO_RXSHORTSYNC	SIO4_IOCTL_USC_BSC_RX_SHORT	For Zilog firmware models.

Values		-1	Retrieve the current setting.
	GSC_ENABLED	SIO4_USC_BSC_SHORT_YES	Option to apply.
	GSC_DISABLED	SIO4_USC_BSC_SHORT_NO	Option to apply.

3.4.9.26. GSC_SIO_TXPREAMBLELEN

Used to set the Tx preamble length for a channel. The monolithic driver uses an IOCTL service for the same purpose.

	GSCAPI	Monolithic	Description
Option	GSC_SIO_TXPREAMBLELEN	SIO4_IOCTL_USC_TX_PREAMBLE_LEN	For Zilog firmware models.

Values		-1	Retrieve the current setting.
	GSC_PREAMBLE_8BITS	SIO4_USC_TX_PREAMBLE_LEN_8_BITS	Option to apply.
	GSC_PREAMBLE_16BITS	SIO4_USC_TX_PREAMBLE_LEN_16_BITS	Option to apply.
	GSC_PREAMBLE_32BITS	SIO4_USC_TX_PREAMBLE_LEN_32_BITS	Option to apply.
	GSC_PREAMBLE_64BITS	SIO4_USC_TX_PREAMBLE_LEN_64_BITS	Option to apply.

3.4.9.27. GSC_SIO_TXPREAMBLEPATTERN

Used to set the Tx preamble pattern for a channel. The monolithic driver uses an IOCTL service for the same purpose.

	GSCAPI	Monolithic	Description
Option	GSC_SIO_TXPREAMBLEPATTERN	SIO4_IOCTL_USC_TX_PREAMBLE_PAT	For Zilog firmware models.

Values		-1	Retrieve the current setting.
	GSC_PREAMBLE_ALL_0	SIO4_USC_TX_PREAMBLE_PAT_0	Option to apply.
	GSC_PREAMBLE_ALL_1	SIO4_USC_TX_PREAMBLE_PAT_1	Option to apply.
	GSC_PREAMBLE_ALL_0_1	SIO4_USC_TX_PREAMBLE_PAT_10	Option to apply.
	GSC_PREAMBLE_ALL_1_0	SIO4_USC_TX_PREAMBLE_PAT_01	Option to apply.

3.4.9.28. GSC_SIO_ORDERING

Used to set the byte and bit order used in bisync16 mode on a channel. This feature is unsupported by the monolithic driver.

3.4.9.29. GSC_SIO_ORDERING

Used to set the byte and bit order used in bisync16 mode on a channel. This feature is unsupported by the monolithic driver.

3.4.9.30. GSC_SIO_MAXRXCOUNT

Used to set the maximum Rx count allowed in bisync16 mode on a channel. This feature is unsupported by the monolithic driver.

3.4.10. GscSio4GetOption

While the GSCAPI has a dedicated function for retrieving current option settings, the monolithic driver uses the same IOCTL function and service as for setting the feature. The difference is that passing in a value of `-1` is almost a universal means of retrieving a current setting.

3.4.11. GscSio4ChannelSetPinMode & GscSio4ChannelSetPinValue

GscSio4ChannelSetPinMode(…) configures a pin to be either its default functionality (via `GSC_PIN_AUTO`) or as GPIO output (via `GSC_PIN_GPIO`). As a GPIO output, GscSio4ChannelSetPinValue(…) sets the output to either an output high (`1`) or an output low (`0`). The monolithic driver supports this same functionality via IOCTL services. Where supported by firmware, each such service supports the option of setting each pin to its default functionality, a GPIO output, either high or low, or reading its current configuration by passing in a value of `-1`. Some of the pins are not configurable and others have additional configuration capabilities. The GSCAPI's set of pins is taken from the `GSC_TOKENS` enumeration, but are limited to those options which are actual pins.

3.4.11.1. GSC_PIN_RX_CLOCK

This is an input only pin and is not configurable.

3.4.11.2. GSC_PIN_RX_DATA

This is an input only pin and is not configurable.

3.4.11.3. GSC_PIN_CTS

This pin can be an input or an output.

	GSCAPI	**Monolithic**	**Description**
Option	`GSC_PIN_CTS`	`SIO4_IOCTL_USC_CTS_CFG`	For Zilog firmware models.

Values		`-1`	Retrieve the current setting.
	`GSC_PIN_AUTO`	`SIO4_USC_CTS_CFG_IN_CBL_CTS`	Option to apply.
	GPIO Output `1`	`SIO4_USC_CTS_CFG_OUT_1`	Option to apply.
	GPIO Output `0`	`SIO4_USC_CTS_CFG_OUT_0`	Option to apply.
		`SIO4_USC_CTS_CFG_TRI`	Option to apply.

3.4.11.4. GSC_PIN_RX_ENV

This is an input only pin and is not configurable.

3.4.11.5. GSC_PIN_DCD

This is an output only pin.

	GSCAPI	Monolithic	Description
Option	`GSC_PIN_DCD`	`SIO4_IOCTL_Z16_CBL_DCD_CFG`	For Zilog firmware models.

Values		`-1`	Retrieve the current setting.
	`GSC_PIN_AUTO`	`SIO4_Z16_CBL_DCD_CFG_OUT_USC_DCD`	Option to apply.
	GPIO Output `1`	`SIO4_Z16_CBL_DCD_CFG_OUT_1`	Option to apply.
	GPIO Output `0`	`SIO4_Z16_CBL_DCD_CFG_OUT_0`	Option to apply.
		`SIO4_Z16_CBL_DCD_CFG_OUT_RTS`	Option to apply.

3.4.11.6. GSC_PIN_TX_CLOCK

This is an output only pin.

	GSCAPI	Monolithic	Description
Option	`GSC_PIN_TX_CLOCK`	`SIO4_IOCTL_Z16_CBL_TXC_CFG`	For Zilog firmware models.

Values		`-1`	Retrieve the current setting.
	`GSC_PIN_AUTO`	`SIO4_Z16_CBL_TXC_CFG_OUT_USC_TXC`	Option to apply.
	GPIO Output `1`	`SIO4_Z16_CBL_TXC_CFG_OUT_1`	Option to apply.
	GPIO Output `0`	`SIO4_Z16_CBL_TXC_CFG_OUT_0`	Option to apply.
		`SIO4_Z16_CBL_TXC_CFG_OUT_CBL_RXA`	Option to apply.
		`SIO4_Z16_CBL_TXC_CFG_OUT_CBL_RXC`	Option to apply.
		`SIO4_Z16_CBL_TXC_CFG_OUT_OSC`	Option to apply.
		`SIO4_Z16_CBL_TXC_CFG_OUT_OSC_INV`	Option to apply.
		`SIO4_Z16_CBL_TXC_CFG_OUT_USC_RXC`	Option to apply.

3.4.11.7. GSC_PIN_TX_DATA

This is an output only pin.

	GSCAPI	Monolithic	Description
Option	`GSC_PIN_TX_DATA`	`SIO4_IOCTL_Z16_CBL_TXD_CFG`	For Zilog firmware models.

Values		`-1`	Retrieve the current setting.
	`GSC_PIN_AUTO`	`SIO4_Z16_CBL_TXD_CFG_OUT_USC_TXD`	Option to apply.
	GPIO Output `1`	`SIO4_Z16_CBL_TXD_CFG_OUT_1`	Option to apply.
	GPIO Output `0`	`SIO4_Z16_CBL_TXD_CFG_OUT_0`	Option to apply.

	GSCAPI	Monolithic	Description
Option	`GSC_PIN_TX_DATA`	`SIO4_IOCTL_SYNC_TXD_CFG`	For -SYNC firmware models.

Values		`-1`	Retrieve the current setting.
	`GSC_PIN_AUTO`	`SIO4_SYNC_TXD_CFG_ACTIVE_HI`	Option to apply.
	GPIO Output `1`	`SIO4_SYNC_TXD_CFG_OUT_1`	Option to apply.
	GPIO Output `0`	`SIO4_SYNC_TXD_CFG_OUT_0`	Option to apply.
		`SIO4_SYNC_TXD_CFG_ACTIVE_LO`	Option to apply.

3.4.11.8. GSC_PIN_RTS

This is an output only pin.

	GSCAPI	Monolithic	Description
Option	`GSC_PIN_RTS`	`SIO4_IOCTL_Z16_CBL_RTS_CFG`	For Zilog firmware models.
Values		`-1`	Retrieve the current setting.
	`GSC_PIN_AUTO`	`SIO4_Z16_CBL_RTS_CFG_OUT_RTS`	Option to apply.
	GPIO Output `1`	`SIO4_Z16_CBL_RTS_CFG_OUT_1`	Option to apply.
	GPIO Output `0`	`SIO4_Z16_CBL_RTS_CFG_OUT_0`	Option to apply.
		`SIO4_Z16_CBL_RTS_CFG_OUT_USC_CTS`	Option to apply.

3.4.11.9. GSC_PIN_TX_ENV

This is an output only pin.

	GSCAPI	Monolithic	Description
Option	`GSC_PIN_TX_ENV`	`SIO4_IOCTL_SYNC_TXE_CFG`	For -SYNC firmware models.
Values		`-1`	Retrieve the current setting.
	`GSC_PIN_AUTO`	`SIO4_SYNC_TXE_CFG_ACTIVE_HI`	Option to apply.
	GPIO Output `1`	`SIO4_SYNC_TXE_CFG_OUT_1`	Option to apply.
	GPIO Output `0`	`SIO4_SYNC_TXE_CFG_OUT_0`	Option to apply.
		`SIO4_SYNC_TXE_CFG_ACTIVE_LO`	Option to apply.

3.4.11.10. GSC_PIN_AUXCLK

This pin is both RxAuxClock, which is always available, and TxAuxClock, which is configurable and can be disabled.

	GSCAPI	Monolithic	Description
Option	`GSC_PIN_AUXCLK`	`SIO4_IOCTL_Z16_CBL_TXAUXC_CFG`	For Zilog firmware models.
Values		`-1`	Retrieve the current setting.
	`GSC_PIN_AUTO`	`SIO4_Z16_CBL_TXAUXC_CFG_OUT_OSC`	Option to apply.
	GPIO Output `1`	`SIO4_Z16_CBL_TXAUXC_CFG_OUT_1`	Option to apply.
	GPIO Output `0`	`SIO4_Z16_CBL_TXAUXC_CFG_OUT_0`	Option to apply.
		`SIO4_Z16_CBL_TXAUXC_CFG_TRI`	Option to apply.

	GSCAPI	Monolithic	Description
Option	`GSC_PIN_AUXCLK`	`SIO4_IOCTL_SYNC_TXAUXC_CFG`	For -SYNC firmware models.
Values		`-1`	Retrieve the current setting.
	`GSC_PIN_AUTO`	`SIO4_SYNC_TXAUXC_CFG_OSC_HALF`	Option to apply.
	GPIO Output `1`	`SIO4_SYNC_TXAUXC_CFG_OUT_1`	Option to apply.
	GPIO Output `0`	`SIO4_SYNC_TXAUXC_CFG_OUT_0`	Option to apply.
		`SIO4_SYNC_TXAUXC_CFG_TRI`	Option to apply.

3.4.12. GscSio4ChannelGetPinMode & GscSio4ChannelGetPinValue

GscSio4ChannelSetPinMode(…) reads a pin’s configuration. When configured as a GPIO, the function GscSio4ChannelGetPinValue(…) reads its current output value. The monolithic driver supports this same functionality via IOCTL services, which are given in the contents of the corresponding Set functions (section 3.4.11, page 41). Additionally, the monolithic driver retrieves each pin’s current configuration/output value by passing a `-1` to the very same IOCTL services.

3.4.13. GscSio4ChannelFifoSizes

GscSio4ChannelFifoSizes(…) returns the size, in bytes, of the channel's Transmit and Receive FIFOs. The monolithic driver uses the `SIO4_IOCTL_QUERY` IOCTL service for the same purpose. In this case, the option value is passed to the service and the result is returned in its place. Also, the GSCAPI encodes the results, while the monolithic driver uses the `s32*` type for all options.

	GSCAPI	Monolithic	Description
Option	`GscSio4ChannelFifoSizes()`	`SIO4_QUERY_FIFO_SIZE_RX` `SIO4_QUERY_FIFO_SIZE_TX`	
Values	`0 – 32,768`	`0 – 32,768`	Value returned.

3.4.14. GscSio4ChannelFifoCounts

GscSio4ChannelFifoCounts(…) returns the current number of bytes in the channel's Transmit and Receive FIFOs. The monolithic driver uses IOCTL services for the same purpose. Also, the GSCAPI encodes the results, while the monolithic driver uses the `s32*` type for all options.

	GSCAPI	Monolithic	Description
Option	`GscSio4ChannelFifoCounts()`	`SIO4_IOCTL_RX_FIFO_FILL_LEVEL` `SIO4_IOCTL_TX_FIFO_FILL_LEVEL`	
Values	`0 – 32,768`	`0 – 32,768`	Value returned.

3.4.15. GscSio4ChannelSetTxAlmost and GscSio4ChannelGetTxAlmost

GscSio4ChannelSetTxAlmost(…) programs the "Almost Full" and "Almost Empty" registers in the Transmit FIFO for a single channel. GscSio4ChannelGetTxAlmost(…) retrieves the settings. The monolithic driver uses IOCTL services for the same purpose. Also, the GSCAPI encodes the settings, while the monolithic driver uses the `s32*` type for all options.

	GSCAPI	Monolithic	Description

Option	`GscSio4ChannelSetTxAlmost()`
	`GscSio4ChannelGetTxAlmost()`
		`SIO4_IOCTL_TX_FIFO_AE`
		`SIO4_IOCTL_TX_FIFO_AF`	

Values		`-1`	Retrieve the current setting.
	`0 – 32,767`	`0 – 32,767`	Option to apply.

3.4.16. GscSio4ChannelSetRxAlmost and GscSio4ChannelGetRxAlmost

GscSio4ChannelSetRxAlmost(…) programs the "Almost Full" and "Almost Empty" registers in the Transmit FIFO for a single channel. GscSio4ChannelGetRxAlmost(…) retrieves the settings. The monolithic driver uses IOCTL services for the same purpose. Also, the GSCAPI encodes the settings, while the monolithic driver uses the `s32*` type for all options.

	GSCAPI	Monolithic	Description

Option	`GscSio4ChannelSetRxAlmost()`
	`GscSio4ChannelGetRxAlmost()`
		`SIO4_IOCTL_RX_FIFO_AE`
		`SIO4_IOCTL_RX_FIFO_AF`	

Values		`-1`	Retrieve the current setting.
	`0 – 32,767`	`0 – 32,767`	Option to apply.

3.4.17. GscSio4ChannelCheckForData

GscSio4ChannelCheckForData(…) determines whether a packet has been received on the specified channel. If a packet has been received, a DMA transfer is initiated to return the data. The monolithic driver itself has no corresponding functionality. However, the HDLC Protocol Library has a blocking call to read a frame. This function may be a suitable substitute, if one is using HDLC.

	GSCAPI	Monolithic	Description
Option	`GscSio4ChannelCheckForData()`	`sio4_hdlc_rx_frame()`	For Zilog firmware models. Requires the HDLC Protocol Library

3.4.18. GscSio4ChannelReceivePacket

GscSio4ChannelReceivePacket(…) determines whether a packet has been received on the specified channel. If a packet has been received, a DMA transfer is initiated to return the data. The monolithic driver itself has no corresponding functionality. The closest approximation is the Rx Frame call from the HDLC Protocol Library, if one is using HDLC.

	GSCAPI	Monolithic	Description
Option	`GscSio4ChannelReceivePacket()`	`sio4_hdlc_rx_frame()`	For Zilog firmware models. Requires the HDLC Protocol Library

3.4.19. GscSio4ChannelReceiveData

GscSio4ChannelReceiveData(…) starts the reception of data on the specified channel. If this function returns before completion, the value pointed to by *id* will contain a unique identifier that can be used to determine the progress of the transfer. The monolithic driver has no equivalent function call for the same purpose. The closest approximation is the API's read call, which is a blocking call.

	GSCAPI	Monolithic	Description
Option	`GscSio4ChannelReceiveData()`	`sio4_read()`	

3.4.20. GscSio4ChannelReceiveDataAndWait

GscSio4ChannelReceiveDataAndWait(…) starts the reception of data on the specified channel. This function will not return until the entire transfer has completed or the timeout period has expired. The monolithic driver uses an equivalent function call for the same purpose.

	GSCAPI	Monolithic	Description
Option	`GscSio4ChannelReceiveDataAndWait()`	`sio4_read()`	

3.4.21. GscSio4ChannelReceivePlxPhysData

GscSio4ChannelReceivePlxPhysData(…) starts the reception of data on the specified channel. The data received on the channel is transferred into the physically contiguous memory buffer pointed to by *buffer*. When this function returns, the value pointed to by *id* will contain a unique identifier that can be used to determine the progress of the transfer. The monolithic driver has no equivalent function call for the same purpose. The closest approximation is the API's read call, which is a blocking call.

	GSCAPI	Monolithic	Description
Option	`GscSio4ChannelReceivePlxPhysData()`	`sio4_read()`	

3.4.22. GscSio4ChannelTransmitData

GscSio4ChannelTransmitData(…) starts the transmission of data on the specified channel. When this function returns, the value pointed to by *id* will contain a unique identifier that can be used to determine the progress of the transfer. The monolithic driver has no equivalent function call for the same purpose. The closest approximation is the API's write call, which is a blocking call.

	GSCAPI	Monolithic	Description
Option	`GscSio4ChannelTransmitData()`	`sio4_write()`	

3.4.23. GscSio4ChannelTransmitDataAndWait

GscSio4ChannelTransmitDataAndWait(…) starts the transmission of data on the specified channel. This function will not return until the entire transfer has completed or the timeout period has expired. The monolithic driver uses an equivalent function call for the same purpose.

	GSCAPI	Monolithic	Description
Option	`GscSio4ChannelTransmitDataAndWait()`	`sio4_write()`	

3.4.24. GscSio4ChannelTransmitPlxPhysData

GscSio4ChannelTransmitPlxPhysData(…) starts the transmission of data on the specified channel. When this function returns, the value pointed to by *id* will contain a unique identifier that can be used to determine the progress of the transfer. The monolithic driver has no equivalent function call for the same purpose. The closest approximation is the API's write call, which is a blocking call.

	GSCAPI	Monolithic	Description
Option	`GscSio4ChannelTransmitPlxPhysData()`	`sio4_write()`	

3.4.25. GscSio4ChannelQueryTransfer

GscSio4ChannelQueryTransfer(…) is used to determine the status of a transfer that was initiated by a call to either GscSio4ChannelReceiveData(…) or GscSio4ChannelTransmitData(…). The monolithic driver has no equivalent functionality. The closest approximation is the driver's Wait Event service in which a thread can wait for an I/O call to complete.

3.4.26. GscSio4ChannelWaitForTransfer

GscSio4ChannelWaitForTransfer(…) is used to wait for the completion of a transfer that was initiated by a call to either GscSio4ChannelReceiveData(…) or GscSio4ChannelTransmitData(…). The routine will return when either the transfer completes or the timeout period expires. The monolithic driver has no equivalent functionality. The closest approximation is the driver's Wait Event service in which a thread can wait for an I/O call to complete.

3.4.27. GscSio4ChannelFlushTransfer

GscSio4ChannelFlushTransfer(…) is used to force any data that is in the Rx FIFO to be transferred via DMA to memory. For a Tx channel, data is transferred to the Tx FIFO until it is full. The monolithic driver has no identical functionality. The equivalent functionality is achieved by repeating the API's read or write call until all data is either received or written.

3.4.28. GscSio4ChannelRemoveTransfer

GscSio4ChannelRemoveTransfer(…) is used to remove a pending transfer from the transfer queue. The monolithic driver has no identical functionality as I/O operations are blocking. The closest similar functionality would be the I/O Abort IOCTL services.

	GSCAPI	Monolithic	Description
Option	`GscSio4ChannelRemoveTransfer()`	`SIO4_IOCTL_RX_IO_ABORT` `SIO4_IOCTL_TX_IO_ABORT`	

3.4.29. GscSio4ChannelRegisterInterrupt

GscSio4ChannelRegisterInterrupt(…) is used register a callback routine with the interrupt handler. The monolithic driver has no corresponding functionality as its Event Notification mechanism doesn't use callbacks (see section 2.34, page 16).

Interrupts and Event Notification:

The GSCAPI interrupts are supported by the monolithic driver's interrupt and Wait Event services. The monolithic driver supports numerous events in addition to those supported by the GSCAPI.

	GSCAPI	Monolithic	Description
Option	`GscSio4ChannelRegisterInterrupt()`	`SIO4_IOCTL_IRQ_GSC_ENABLE` Wait Event Services	

Values	`GSC_INTR_SYNC_DETECT`	`SIO4_IRQ_SYNC_BYTE` `SIO4_WAIT_GSC_SYNC_BYTE`	
	`GSC_INTR_USC`	`SIO4_IRQ_USC` `SIO4_WAIT_GSC_USC`	
	`GSC_INTR_TX_FIFO_EMPTY`	`SIO4_IRQ_TX_FIFO_E` `SIO4_WAIT_GSC_TX_FIFO_E`	
	`GSC_INTR_TX_FIFO_FULL`	`SIO4_IRQ_TX_FIFO_F` `SIO4_WAIT_GSC_TX_FIFO_F`	
	`GSC_INTR_TX_FIFO_ALMOST_EMPTY`	`SIO4_IRQ_TX_FIFO_AE` `SIO4_WAIT_GSC_TX_FIFO_AE`	
	`GSC_INTR_RX_FIFO_EMPTY`	`SIO4_IRQ_RX_FIFO_E` `SIO4_WAIT_GSC_RX_FIFO_E`	
	`GSC_INTR_RX_FIFO_FULL`	`SIO4_IRQ_RX_FIFO_F` `SIO4_WAIT_GSC_RX_FIFO_F`	
	`GSC_INTR_RX_FIFO_ALMOST_FULL`	`SIO4_IRQ_RX_FIFO_AF` `SIO4_WAIT_GSC_RX_FIFO_AF`	
	`GSC_INTR_RX_ENVELOPE`	`SIO4_IRQ_RX_ENV` `SIO4_WAIT_GSC_RX_ENV`	

Interrupt Configuration:

The GSCAPI and the monolithic drivers support similar interrupt configuration interfaces. The GSCAPI uses the referenced function call and the monolithic driver uses an IOCTL service.

	GSCAPI	Monolithic	Description
Option	`GscSio4ChannelRegisterInterrupt()`	`SIO4_IOCTL_IRQ_GSC_CFG_HIGH`	

Values	`GSC_RISING_EDGE`	If a bit is set, then the corresponding interrupt is configured for rising edge or level high detection.	
	`GSC_FALLING_EDGE`	If a bit is clear, then the corresponding interrupt is configured for falling edge or level low detection.	

3.4.30. GscSio4ChannelSetClock

GscSio4ChannelSetClock(…) is used to set the serial bitrate (baud rate) for a specific channel. The monolithic driver doesn't contain the functionality for deriving the configuration needed to achieve application desired Tx and Rx bitrates. For the most part, the code for these calculations is contained in the Protocol Libraries as those calculations are dependent on the protocol in use and possibly on how the Tx and Rx cable clock signals may be used.

3.5. Protocol Level Routines

Both drivers provide protocol specific support, but not for the same set of protocols. Where a protocol is supported by both, the implementations are entirely different. Refer to the monolithic driver's appropriate protocol specific reference manual.

3.5.1. GscSio4HdlcGetDefaults

GscSio4HdlcGetDefaults(…) returns the default HDLC configuration structure. The monolithic driver provides similar functionality, but requires some information from the caller and provides different information to the caller.

	GSCAPI	**Monolithic**	**Description**
Option	`GscSio4HdlcGetDefaults()`	`sio4_hdlc_init_data()`	

3.5.2. GscSio4HdlcSetConfig and GscSio4HdlcGetConfig

GscSio4HdlcSetConfig(…) sets the mode of the specified channel to HDLC and sets the current configuration to the values specified in the *config* parameter. GscSio4HdlcGetConfig(…) retrieves the current configuration from the channel. The monolithic driver provides similar functionality, but uses an entirely different structure.

	GSCAPI	**Monolithic**	**Description**
Options	`GscSio4HdlcSetConfig()`	`sio4_hdlc_set()`	
	`GscSio4HdlcGetConfig()`	`sio4_hdlc_get()`	

3.5.3. GscSio4AsyncGetDefaults

GscSio4AsyncGetDefaults(…) returns the default HDLC configuration structure. The monolithic driver provides similar functionality, but requires some information from the caller and provides different information to the caller.

	GSCAPI	**Monolithic**	**Description**
Option	`GscSio4AsyncGetDefaults()`	`sio4_async_init_data()`	

3.5.4. GscSio4AsyncSetConfig and GscSio4AsyncGetConfig

GscSio4AsyncSetConfig(…) sets the mode of the specified channel to Asynchronous and sets the current configuration to the values specified in the *config* parameter. GscSio4AsyncGetConfig(…) retrieves the current configuration from the channel. The monolithic driver provides similar functionality, but uses an entirely different structure.

	GSCAPI	Monolithic	Description
Options	`GscSio4AsyncSetConfig()`	`sio4_async_set()`	
	`GscSio4AsyncGetConfig()`	`sio4_async_get()`	

3.5.5. GscSio4BiSyncGetDefaults

GscSio4BiSyncGetDefaults(…) returns the default BiSync configuration structure. The monolithic driver does not provide dedicated BiSync Protocol support.

3.5.6. GscSio4BiSyncSetConfig and GscSio4BiSyncGetConfig

GscSio4BiSyncSetConfig(…) sets the mode of the specified channel to BiSync and sets the current configuration to the values specified in the *config* parameter. GscSio4BiSyncGetConfig(…) retrieves the current configuration from the channel. The monolithic driver does not provide dedicated BiSync Protocol support.

3.5.7. GscSio4SyncGetDefaults

GscSio4SyncGetDefaults(…) returns the default Sync configuration structure. The monolithic driver provides SYNC protocol support, but that support doesn't provide similar functionality.

3.5.8. GscSio4SyncSetConfig and GscSio4SyncGetConfig

GscSio4SyncSetConfig(…) sets the mode of the specified channel to Sync and sets the current configuration to the values specified in the *config* parameter. GscSio4SyncGetConfig(…) retrieves the current configuration from the channel. The monolithic driver provides similar functionality, but uses an entirely different structures and multiple functions.

	GSCAPI	Monolithic	Description

Options	`GscSio4SyncSetConfig()`	`sio4_sync_set()`
		`sio4_sync_rx_set()`
		`sio4_sync_tx_set()`	

	`GscSio4SyncGetConfig()`	`sio4_sync_get()`
		`sio4_sync_rx_get()`
		`sio4_sync_tx_get()`	

3.5.9. GscSio4BiSync16GetDefaults

GscSio4BiSync16GetDefaults(…) returns the default bisync16 configuration structure. The monolithic driver doesn't provide any BiSync16 specific protocol support.

3.5.10. GscSio4BiSync16SetConfig and GscSio4BiSync16GetConfig

GscSio4BiSync16SetConfig(…) sets the mode of the specified channel to Sync and sets the current configuration to the values specified in the *config* parameter. GscSio4BiSync16GetConfig(…) retrieves the current configuration from the channel. The monolithic driver doesn't provide any BiSync16 specific protocol support.

3.5.11. GscSio4BiSync16GetTxCounts

GscSio4BiSync16GetTxCounts(…) is used to retrieve the initial and remaining Tx counts for a channel configured in bisync16 mode. The monolithic driver doesn't provide any BiSync16 specific protocol support.

3.5.12. GscSio4BiSync16GetRxCounts

GscSio4BiSync16GetRxCounts(…) is used to retrieve the initial and remaining Rx counts for a channel configured in bisync16 mode. The monolithic driver doesn't provide any BiSync16 specific protocol support.

3.5.13. GscSio4BiSync16EnterHuntMode

GscSio4BiSync16EnterHuntMode(…) is used to cause a channel configured in bisync16 mode to enter hunt mode. The monolithic driver doesn't provide any BiSync16 specific protocol support.

3.5.14. GscSio4BiSync16AbortTx

GscSio4BiSync16AbortTx(…) is used to cause a channel configured in bisync16 mode to abort the current transmission. The monolithic driver doesn't provide any BiSync16 specific protocol support.

3.5.15. GscSio4BiSync16Pause

GscSio4BiSync16Pause(…) is used to cause a channel configured in bisync16 mode to pause the current transmission. The monolithic driver doesn't provide any BiSync16 specific protocol support.

3.5.16. GscSio4BiSync16Resume

GscSio4BiSync16Resume(…) is used to cause a channel configured in bisync16 mode to pause the current transmission. The monolithic driver doesn't provide any BiSync16 specific protocol support.

3.6. CTC Protocol Routines

The GSCAPI provide CTC Protocol support. The monolithic driver does not.

3.6.1. GscSio4CTCAddMajorFrame

GscSio4CTCAddMajorFrame(…) adds a CTC major frame to the TX SRAM for the specified channel. The monolithic driver doesn't provide any CTC protocol support.

3.6.2. GscSio4CTCAddMinorFrame

GscSio4CTCAddMinorFrame(…) adds a CTC minor frame to the TX SRAM for the specified channel. The monolithic driver doesn't provide any CTC protocol support.

3.6.3. GscSio4CTCGetActiveMajorFrame

GscSio4CTCGetActiveMajorFrame(…) returns the frame number of the CTC major frame actively being transmitted. The monolithic driver doesn't provide any CTC protocol support.

3.6.4. GscSio4GetActiveMajorFrame

GscSio4GetActiveMajorFrame(…) returns the current configuration structure for a single channel that is set to CTC mode. The monolithic driver doesn't provide any CTC protocol support.

3.6.5. GscSio4CTCGetDefaults

GscSio4CTCGetDefaults(…) returns the default CTC configuration structure. The monolithic driver doesn't provide any CTC protocol support.

3.6.6. GscSio4CTCReceiveFrames

GscSio4CTCReceiveFrames(…) returns the frame number of the CTC major frame actively being transmitted. The monolithic driver doesn't provide any CTC protocol support.

3.6.7. GscSio4CTCResetTimer

GscSio4CTCResetTimer(…) resets the CTC timer value. The monolithic driver doesn’t provide any CTC protocol support.

3.6.8. GscSio4CTCSetConfig

GscSio4CTCSetConfig(…) sets the current configuration structure for a single CTC channel. The monolithic driver doesn’t provide any CTC protocol support.

3.6.9. GscSio4CTCSetTimer

GscSio4CTCSetTimer(…) sets the CTC timer BCD or binary value. The monolithic driver doesn’t provide any CTC protocol support.

3.6.10. GscSio4CTCSwitchMajorFrame

GscSio4CTCSwitchMajorFrame(…) adds a CTC major frame to TX SRAM for the specified channel and hands off active CTC transmission to the added major frame, causing the previously transmitting major frame to go dormant. The monolithic driver doesn’t provide any CTC protocol support.

3.6.11. GscSio4CTCTransmitFrames

GscSio4CTCTransmitFrames(…) transmits a loop of CTC minor frames on the specified channel. The monolithic driver doesn’t provide any CTC protocol support.

Document History

Revision	**Description**
March 19, 2024	Updated the release date.
November 16, 2023	Updated the release date. Minor editorial changes.
June 15, 2023	Updated the release date. Minor editorial updates.
December 13, 2022	Updated the release date. Minor editorial updates.
September 26, 2022	Initial release.