ADADIO/2

8 A/D Channels, 4 D/A Channels, 16-bit
With 8-bit Discrete Digital I/O

All Form Factors
…-ADADIO
…-ADADIO2

Linux Device Driver And API Library User Manual

Manual Revision: January 30, 2023
Driver Release Version 4.9.102.44.0

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

Preface

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

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

Table of Contents

1. Introduction 7

1.1. Purpose 7

1.2. Acronyms 7

1.3. Definitions 7

1.4. Software Overview 7

1.4.1. Basic Software Architecture 7

1.4.2. API Library 8

1.4.3. Device Driver 8

1.5. Hardware Overview 8

1.6. Reference Material 8

1.7. Licensing 9

2. Installation 10

2.1. CPU and Kernel Support 10

2.1.1. 32-bit Support Under 64-bit Environments 11

2.2. The /proc/ File System 11

2.3. File List 11

2.4. Directory Structure 11

2.5. Installation 12

2.6. Removal 12

2.7. Overall Make Script 12

2.8. Environment Variables 13

2.8.1. `GSC_API_COMP_FLAGS` 13

2.8.2. `GSC_API_LINK_FLAGS` 13

2.8.3. `GSC_LIB_COMP_FLAGS` 13

2.8.4. `GSC_LIB_LINK_FLAGS` 14

2.8.5. `GSC_APP_COMP_FLAGS` 14

2.8.6. `GSC_APP_LINK_FLAGS` 14

3. Main Interface Files 15

3.1. Main Header File 15

3.2. Main Library File 15

3.2.1. Build 15

3.2.2. System Libraries 16

4. API Library 17

4.1. Files 17

4.2. Build 17

4.3. Library Use 17

4.4. Macros 18

4.4.1. IOCTL 18
4.4.2. Registers 18
4.5. Data Types 18
4.6. Functions 18
4.6.1. adadio_close() 19
4.6.2. adadio_init() 19
4.6.3. adadio_ioctl() 20
4.6.4. adadio_open() 21
4.6.5. adadio_read() 22
4.6.6. ADADIO Write 23
4.7. IOCTL Services 23
4.7.1. ADADIO_IOCTL_AIN_BUF_CLEAR 23
4.7.2. ADADIO_IOCTL_AIN_BUF_ENABLE 23
4.7.3. ADADIO_IOCTL_AIN_BUF_SIZE 23
4.7.4. ADADIO_IOCTL_AIN_BUF_STS 24
4.7.5. ADADIO_IOCTL_AIN_CHAN_LAST 24
4.7.6. ADADIO_IOCTL_AIN_MODE 25
4.7.7. ADADIO_IOCTL_AIN_NRATE 25
4.7.8. ADADIO_IOCTL_AIN_TRIGGER 26
4.7.9. ADADIO_IOCTL_AOUT_CH_*X*_WRITE 26
4.7.10. ADADIO_IOCTL_AOUT_ENABLE 26
4.7.11. ADADIO_IOCTL_AOUT_STROBE 27
4.7.12. ADADIO_IOCTL_AOUT_STROBE_ENABLE 27
4.7.13. ADADIO_IOCTL_AUTO_CALIBRATE 27
4.7.14. ADADIO_IOCTL_DATA_FORMAT 27
4.7.15. ADADIO_IOCTL_DIO_PIN_READ 28
4.7.16. ADADIO_IOCTL_DIO_PIN_WRITE 28
4.7.17. ADADIO_IOCTL_DIO_PORT_DIR 28
4.7.18. ADADIO_IOCTL_DIO_PORT_READ 29
4.7.19. ADADIO_IOCTL_DIO_PORT_WRITE 29
4.7.20. ADADIO_IOCTL_INITIALIZE 29
4.7.21. ADADIO_IOCTL_IRQ_SEL 30
4.7.22. ADADIO_IOCTL_LOOPBACK_CHANNEL 30
4.7.23. ADADIO_IOCTL_QUERY 30
4.7.24. ADADIO_IOCTL_REG_MOD 31
4.7.25. ADADIO_IOCTL_REG_READ 32
4.7.26. ADADIO_IOCTL_REG_WRITE 32
4.7.27. ADADIO_IOCTL_RX_IO_ABORT 33
4.7.28. ADADIO_IOCTL_RX_IO_MODE 33
4.7.29. ADADIO_IOCTL_RX_IO_TIMEOUT 33
4.7.30. ADADIO_IOCTL_WAIT_CANCEL 34
4.7.31. ADADIO_IOCTL_WAIT_EVENT 35
4.7.32. ADADIO_IOCTL_WAIT_STATUS 36

5. The Driver 38

5.1. Files 38
5.2. Build 38
5.3. Startup 38
5.3.1. Manual Driver Startup Procedures 38
5.3.2. Automatic Driver Startup Procedures 39
5.4. Verification 41

5.5. Version 41
5.6. Shutdown 41

6. Document Source Code Examples 42

6.1. Files 42
6.2. Build 42
6.3. Library Use 42

7. Utility Source Code 43

7.1. Files 43
7.2. Build 43
7.3. Library Use 43

8. Operating Information 44

8.1. Debugging Aids 44
8.1.1. Device Identification 44
8.1.2. Detailed Register Dump 44
8.2. Analog Input Configuration 44
8.3. I/O Modes 44
8.3.1. PIO - Programmed I/O 44
8.3.2. BMDMA - Block Mode DMA 45
8.3.3. DMDMA - Demand Mode DMA 45

9. Sample Applications 46

9.1. aout - Analog Output - …/aout/ 46
9.2. din - Digital Input - …/din/ 46
9.3. dout - Digital Output - …/dout/ 46
9.4. id - Identify Board - …/id/ 46
9.5. regs - Register Access - …/regs/ 46
9.6. rxrate - Receive Rate - …/rxrate/ 46
9.7. savedata - Save Acquired Data - …/savedata/ 46
9.8. sbtest - Single Board Test - …/sbtest/ 46

Document History 47

Table of Figures

Figure 1 The basic software architecture of Linux based ADADIO applications. 8

1. Introduction

1.1. Purpose

The purpose of this document is to describe the interface to the ADADIO API Library and to the underlying Linux device driver. The API Library software provides the interface between "Application Software" and the device driver. The driver software provides the interface between the API Library and the actual ADADIO hardware. The API Library and driver interfaces are based on the board's functionality.

1.2. Acronyms

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

Acronyms	Description
API	Application Programming Interface
BMDMA	Block Mode DMA
DMA	Direct Memory Access
DMDMA	Demand Mode DMA
GSC	General Standards Corporation
PC104P	This refers to the PC/104+ form factor.
PCI	Peripheral Component Interconnect
PIO	Programmed I/O
PMC	PCI Mezzanine Card
PMC66	This refers to a PMC device capable of operating at 66MHz bus speeds.

1.3. Definitions

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

Term	Definition
…	This is a shortcut representation of the ADADIO installation directory or any of its subdirectories.
ADADIO	This is used as a general reference to any board supported by this driver, which includes the ADADIO and the ADADIO2 model boards.
API Library	This is a library that provides application-level access to ADADIO hardware.
Application	This is a user mode process, which runs in user space with user mode privileges.
Driver	This is the ADADIO device driver, which runs in kernel space with kernel mode privileges.
Library	This is usually a general reference to the API Library.

1.4. Software Overview

1.4.1. Basic Software Architecture

This section describes the general architecture for the basic components that comprise ADADIO applications. The overall architecture is illustrated in Figure 1 below.

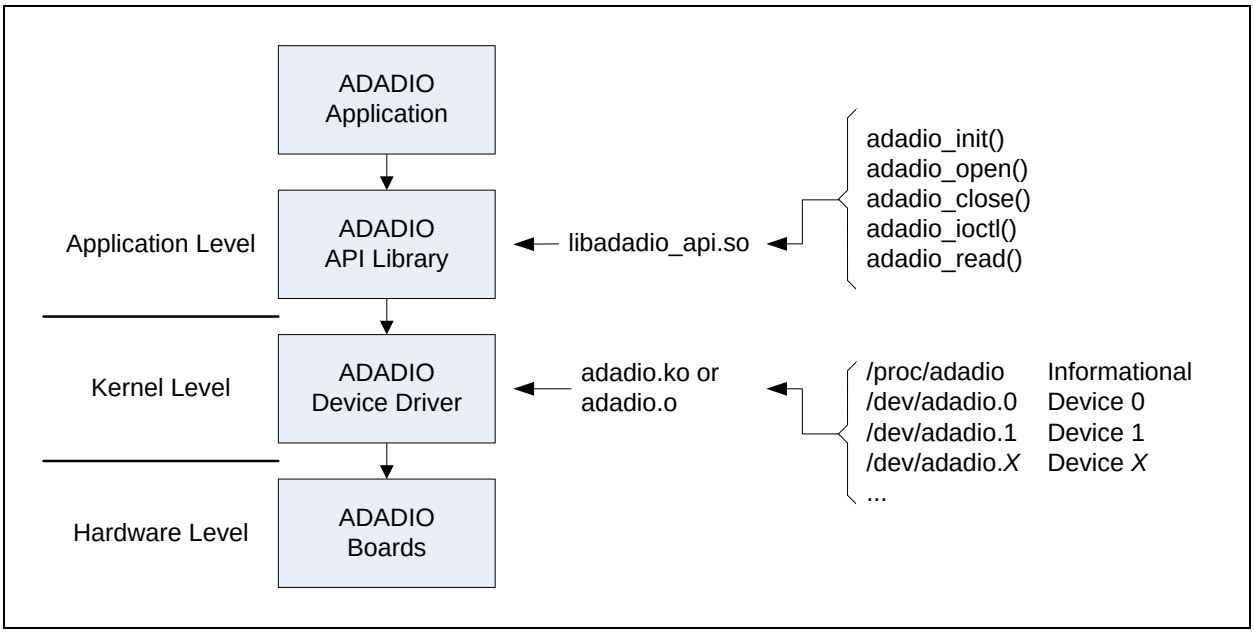

Figure 1 The basic software architecture of Linux based ADADIO applications.

1.4.2. API Library

The primary means of accessing ADADIO boards is via the ADADIO API Library. This library forms a thin layer between the application and the driver. Additional information is given in section 4 beginning on page 17. With the library, applications are able to open and close a device and, while open, perform I/O control and read operations.

1.4.3. Device Driver

The device driver is the host software that provides a means of communicating directly with ADADIO hardware. The driver executes under control of the operating system and runs in Kernel Mode as a Kernel Mode device driver. The driver is implemented as a standard dynamically loadable Linux device driver written in the C programming language. While applications can access the driver directly without use of the API Library, it is recommended that all access is made through the library.

1.5. Hardware Overview

The ADADIO is a high-performance 16-bit analog-to-digital and digital-to-analog I/O interface board. The host side connection is 32-bit PCI based. The external I/O interface varies per model ordered. The board contains eight synchronous 16-bit analog-to-digital input channels capable of performing up to 200,000 conversions per second per channel. All channels are clocked simultaneously and may be synchronized with external equipment either by the ADADIO itself or by an external device. Conversions can be performed on demand or continuously. An onboard receive FIFO of 32k samples collects the converted data for subsequent retrieval by the host. The FIFO allows the ADADIO to buffer data between the cable interface and the PCI bus while maintaining continuous conversions on the cable interface (at least up to the depth of the FIFOs) independent of the PCI bus interface. Converted data can be retrieved using either PIO or DMA. The board also contains four independent asynchronous 16-bit digital-to-analog output channels. In addition, the board includes TTL level digital I/O lines. This consists of an 8-bit bidirectional discrete digital I/O port with one dedicated input and one dedicated output.

1.6. Reference Material

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

- The applicable *ADADIO User Manual* from General Standards Corporation.
- The applicable *ADADIO2 User Manual* from General Standards Corporation.
- The PCI9056 PCI Bus Master Interface Chip data handbook from PLX Technology, Inc. *
- The PCI9080 PCI Bus Master Interface Chip data handbook from PLX Technology, Inc. *

* PLX Technology Inc.
870 Maude Avenue
Sunnyvale, California 94085 USA
Phone: 1-800-759-3735
WEB: http://www.plxtech.com

1.7. Licensing

For licensing information please refer to the text file `LICENSE.txt` in the root installation directory.

2. Installation

2.1. CPU and Kernel Support

The driver is designed to operate with Linux kernel versions 5.x, 4.x, 3.x, 2.6, 2.4 and 2.2 running on a PC system with one or more x86 processors. This release of the driver supports the below listed kernels.

Kernel	Distribution
5.14.10	Red Hat Fedora Core 35
5.11.12	Red Hat Fedora Core 34
5.8.15	Red Hat Fedora Core 33
5.6.6	Red Hat Fedora Core 32
5.3.7	Red Hat Fedora Core 31
5.0.9	Red Hat Fedora Core 30
4.18.16	Red Hat Fedora Core 29
4.16.3	Red Hat Fedora Core 28
4.13.9	Red Hat Fedora Core 27
4.11.8	Red Hat Fedora Core 26
4.8.6	Red Hat Fedora Core 25
4.5.5	Red Hat Fedora Core 24
4.2.3	Red Hat Fedora Core 23
4.0.4	Red Hat Fedora Core 22
3.17.4	Red Hat Fedora Core 21
3.11.10	Red Hat Fedora Core 20
3.9.5	Red Hat Fedora Core 19
3.6.10	Red Hat Fedora Core 18
3.3.4	Red Hat Fedora Core 17
3.1.0	Red Hat Fedora Core 16
2.6.38	Red Hat Fedora Core 15
2.6.35	Red Hat Fedora Core 14
2.6.33	Red Hat Fedora Core 13
2.6.31	Red Hat Fedora Core 12
2.6.29	Red Hat Fedora Core 11
2.6.27	Red Hat Fedora Core 10
2.6.25	Red Hat Fedora Core 9
2.6.23	Red Hat Fedora Core 8
2.6.21	Red Hat Fedora Core 7
2.6.18	Red Hat Fedora Core 6
2.6.15	Red Hat Fedora Core 5
2.6.11	Red Hat Fedora Core 4
2.6.9	Red Hat Fedora Core 3

NOTE: Some older kernel versions are supported (the sources are maintained), but are not tested.

NOTE: While only Red Hat Fedora distributions are listed, numerous other distributions are supported and have been tested on an as needed basis.

NOTE: The driver will have to be built before being used as it is provided in source form only.

NOTE: The driver has not been tested with a non-versioned kernel.

NOTE: The driver has not been tested for SMP operation.

2.1.1. 32-bit Support Under 64-bit Environments

This driver supports 32-bit applications under 64-bit environments. The availability of this feature in the kernel depends on a 64-bit kernel being configured to support 32-bit application compatibility. Additionally, 2.6 kernels prior to 2.6.11 implemented 32-bit compatibility in a way that resulted in some drivers not being able to take advantage of the feature. (In these kernels a driver's IOCTL command codes must be globally unique. Beginning with 2.6.11 this requirement has been lifted.) If the driver is not able to provide 32-bit support under a 64-bit kernel, the "`32-bit support`" field in the `/proc/adadio` file will be "`no`".

2.2. The /proc/ File System

While the driver is running, the text file `/proc/adadio` can be read to obtain information about the driver. Each file entry includes an entry name followed immediately by a colon, a space character, and the entry value. Below is an example of what appears in the file, followed by descriptions of each entry.

```
version: 4.9.102.44
32-bit support: yes
boards: 2
models: ADADIO,ADADIO2
```

Entry	Description
`version`	This gives the driver version number in the form `x.x.x.x`.
`32-bit support`	This reports the driver's support for 32-bit applications. This will be either "`yes`" or "`no`" for 64-bit driver builds and "`yes (native)`" for 32-bit builds.
`boards`	This identifies the total number of boards the driver detected.
`models`	This gives a comma separated list of the device models identified by the driver. One model will be listed for each device identified in the system. For this driver the model numbers listed will be either "`ADADIO`" or "`ADADIO2`."

2.3. File List

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

File	Description
`adadio.linux.tar.gz`	This archive contains the driver, the API Library and all related files.
`adadio_linux_um.pdf`	This is a PDF version of this user manual, which is included in the archive.

2.4. Directory Structure

The following table describes the directory structure utilized by the installed files. During installation the directory structure is created and populated with the respective files.

Directory	Content
`adadio/`	This is the driver root directory. It contains the documentation, the Overall Make Script (section 2.7, page 12) and the below listed subdirectories.
`…/api/`	This directory contains the ADADIO API Library (section 4, page 17).
`…/docsrc/`	This directory contains the code samples from this document (section 6, page 42).
`…/driver/`	This directory contains the driver and its sources (section 5, page 38).
`…/include/`	This directory contains the include files for the various libraries.
`…/lib/`	This directory contains all of the libraries built from the driver archive.
`…/samples/`	This directory contains the sample applications (section 9, page 46).

…/utils/	This directory contains utility sources used by the sample applications (section 7, page 43).

2.5. Installation

Perform installation following the below listed steps. This installs the device driver, the API Library and all related sources and documentation.

1. Create and change to the directory where the files are to be installed, such as `/usr/src/linux/drivers/`. (The path name may vary among distributions and kernel versions.)

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

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

   ```
   tar –xzvf adadio.linux.tar.gz
   ```

2.6. Removal

Perform removal following the below listed steps. This removes the device driver, the API Library and all related sources and documentation.

> **NOTE**: The following steps may require elevated privileges.

1. Shutdown the driver as described in section 5.6 on page 41.

2. Change to the directory where the driver archive was installed, which may have been `/usr/src/linux/drivers/`. (The path name may vary among distributions and kernel versions.)

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

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

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

   ```
   rm –f /dev/adadio.*
   ```

5. If the automated startup procedure was adopted (section 5.3.2, page 39), then edit the system startup script `rc.local` and remove the line that invokes the ADADIO's `start` script. The file `rc.local` should be located in the `/etc/rc.d/` directory.

2.7. Overall Make Script

An Overall Make Script is included in the root installation directory. Executing this script will perform a make for all build targets included in the release. The script also loads the driver and copies the API Library to `/usr/lib/`. The script is named `make_all`. Follow the below steps to perform an overall make.

> **NOTE**: The following steps may require elevated privileges.

1. Change to the driver root directory (`…/adadio/`).

2. Issue the following command to remove all archive build targets.

   ```
   ./make_all clean
   ```

3. Issue the following command to make all archive targets and to load the driver.

```
./make_all
```

> **NOTE**: After the device driver is built the script starts the driver. After building the API Library it is copied to `/usr/lib/`. The script can also perform a clean operation by adding the term "`clean`" as a command line argument. A clean operation does not unload the driver. However, a clean does delete the API Library file copied to `/usr/lib/`.

2.8. Environment Variables

Some build environments may require compiler or linker options not present in the provided make files. To accommodate local environment specific requirements, the provided make files incorporate support for the following set of GSC specific environment variables.

2.8.1. GSC_API_COMP_FLAGS

This environment variable accommodates adding compiler command line options when compiling source files for the API Library. The compiler used by the API Library make file is "`gcc`". The content of this environment variable is noted in the make file's output to the screen. The table below shows a portion of the screen output. The "`xxx`" in the table refers to the contents of the environment variable. This environment variable has no effect on compiling any other distributed source files or linking of any object files.

Undefined or Empty	`== Compiling: init.c`
	`== Compiling: ioctl.c`
	`== Compiling: open.c`

Defined and Not Empty	`== Compiling: init.c (added 'xxx')`
	`== Compiling: ioctl.c (added 'xxx')`
	`== Compiling: open.c (added 'xxx')`

2.8.2. GSC_API_LINK_FLAGS

This environment variable accommodates adding linker command line options when linking object files for the API Library. The linker used by the API Library make file is "`ld`". The content of this environment variable is noted in the make file's output to the screen. The table below shows a portion of the screen output. The "`xxx`" in the table refers to the contents of the environment variable. This environment variable has no effect on compiling of any source files or linking of any other object files.

Undefined or Empty	`==== Linking: ../lib/libadadio_api.so`
Defined and Not Empty	`==== Linking: ../lib/libadadio_api.so (added 'xxx')`

2.8.3. GSC_LIB_COMP_FLAGS

This environment variable accommodates adding compiler command line options when compiling source files for the utility libraries. The compiler used by the utility library make files is "`gcc`". The content of this environment variable is noted in the make files' output to the screen. The table below shows a portion of the screen output. The "`xxx`" in the table refers to the contents of the environment variable. This environment variable has no effect on compiling any other distributed source files or linking of any object files.

Undefined or Empty	`== Compiling: close.c`
	`== Compiling: init.c`
	`== Compiling: ioctl.c`

Defined and Not Empty	`== Compiling: close.c (added 'xxx')`
	`== Compiling: init.c (added 'xxx')`
	`== Compiling: ioctl.c (added 'xxx')`

2.8.4. GSC_LIB_LINK_FLAGS

This environment variable accommodates adding linker command line options when linking object files for the utility libraries. The linker used by the utility library make files is "`ld`". The content of this environment variable is noted in the make files' output to the screen. The table below shows a portion of the screen output. The "`xxx`" in the table refers to the contents of the environment variable. This environment variable has no effect on compiling of any source files or linking of any other object files.

Undefined or Empty	`==== Linking: ../lib/adadio_utils.a`
Defined and Not Empty	`==== Linking: ../lib/adadio_utils.a (added 'xxx')`

2.8.5. GSC_APP_COMP_FLAGS

This environment variable accommodates adding compiler command line options when compiling source files for the sample applications. The compiler used by the sample application make files is "`gcc`". The content of this environment variable is noted in the make files' output to the screen. The table below shows a portion of the screen output. The "`xxx`" in the table refers to the contents of the environment variable. This environment variable has no effect on compiling any other distributed source files or linking of any object files.

Undefined or Empty	`== Compiling: main.c` `== Compiling: perform.c`
Defined and Not Empty	`== Compiling: main.c (added 'xxx')` `== Compiling: perform.c (added 'xxx')`

2.8.6. GSC_APP_LINK_FLAGS

This environment variable accommodates adding linker command line options when linking object files for the sample applications. The linker used by the sample application make files is "`gcc`". The content of this environment variable is noted in the make files' output to the screen. The table below shows a portion of the screen output. The "`xxx`" in the table refers to the contents of the environment variable. This environment variable has no effect on compiling of any source files or linking of any other object files.

Undefined or Empty	`==== Linking: id`
Defined and Not Empty	`==== Linking: id (added 'xxx')`

3. Main Interface Files

This section gives general information on the suggested device interface files to use when developing ADADIO based applications.

3.1. Main Header File

Throughout the remainder of this document references are made to various header files included as part of the ADADIO driver archive. For ease of use it is suggested that applications include only the single header file shown below rather than individually including those headers identified separately later in this document. Including this header file pulls in all other pertinent ADADIO specific header files. Therefore, sources may include only this one ADADIO header and make files may reference only this one ADADIO include directory.

Description	File	Location
Header File	`adadio_main.h`	`…/include/`

3.2. Main Library File

Throughout the remainder of this document references are made to various statically linkable libraries included as part of the ADADIO driver archive. For ease of use it is suggested that applications link only the single library file shown below rather than individually linking those libraries identified separately later in this document. Linking this library file pulls in all other pertinent ADADIO specific static libraries. Therefore, make files may reference only this one ADADIO static library and only this one ADADIO library directory.

Description	File (see note below)	Location
Static Library	`adadio_main.a *`	`…/lib/`
Static Library	`adadio_multi.a *`	

> **NOTE**: The ADADIO API Library is implemented as a shared library and is thus not linked with the ADADIO Main Library. The API Library must be linked with applications by adding the argument `-ladadio_api` to the linker command line.
>
> **NOTE**: For applications using the ADADIO and no other GSC devices, link the `adadio_main.a` library. For applications using multiple GSC device types, link the `xxxx_main.a` library for one of the devices and the `xxxx_multi.a` library for the others. Linking multiple `xxxx_main.a` libraries may likely produce link errors due to duplicate symbols being defined. While it may make little or no difference, it is recommended that one choose the `xxxx_main.a` library from the driver with the largest number in positions three (x.x.X.x.x) and/or four (x.x.x.X.x) in the driver release version number.

3.2.1. Build

The main library is built via the Overall Make Script (section 2.7, page 12). However, the main library can be built separately following the below steps.

1. Change to the directory where the main library resides (`…/lib/`).

2. Remove existing build targets using the below command.

   ```
   make clean
   ```

3. Rebuild the main library by issuing the below command.

   ```
   make
   ```

3.2.2. System Libraries

In addition to linking the static library named above, applications may need to also link in additional system libraries as noted below.

Library	gcc Link Flag
Math	`-lm`
POSIX Thread	`-lpthread`
Real Time	`-lrt`

4. API Library

The ADADIO API Library is the software interface between user applications and the ADADIO device driver. The interface is accessed by including the header file `adadio_api.h`.

> **NOTE:** Contact General Standards Corporation if additional library functionality is required.

4.1. Files

The library files are summarized in the table below.

File	Description
`api/*.c`	These are library source files.
`api/*.h`	These are library header files.
`api/makefile`	This is the library make file.
`api/makefile.dep`	This is an automatically generated make dependency file.
`include/adadio_api.h`	This is the library interface header file.
`lib/libadadio_api.so`	This is the API Library shared library file. *

* The shared library is automatically copied to `/usr/lib/` when it is built.

4.2. Build

The API Library is built via the Overall Make Script (section 2.7, page 12), but can be built separately following the below steps.

> **NOTE**: The API Library shared library is copied to `/usr/lib/`. Therefore, these steps may require elevated privileges.

1. Change to the directory where the library sources are installed (…`/api/`).

2. Remove existing build targets using the below command.

   ```
   make clean
   ```

3. Compile the source files and build the library by issuing the below command.

   ```
   make
   ```

4.3. Library Use

The library is used at application compile time, at application link time and at application run time. At compile time include the below listed header file in each source file using a component of the library interface. At link time include the below listed linker argument on the linker command line. At link time and at run time the library is found in the directory `/usr/lib/`. (The shared library file is automatically copied to `/usr/lib/` when the library is built.)

Description	File	Location	Linker Argument
Header File	`adadio_api.h`	…`/include/`	
Shared Library	`libadadio_api.so`	…`/lib/`	
		`/usr/lib/`	`-ladadio_api`

4.4. Macros

The API Library and driver interfaces include the following macros, which are defined in `adadio.h`.

4.4.1. IOCTL

The IOCTL macros are documented in section 4.7 beginning on page 23.

4.4.2. Registers

The following gives the complete set of ADADIO registers.

4.4.2.1. GSC Registers

The following table gives the complete set of GSC specific ADADIO registers. For detailed definitions of these registers refer to the relevant ADADIO User Manual. Please note that the set of registers supported by any given board may vary according to model and firmware version. For the set of supported registers and detailed definitions of these registers please refer to the appropriate *ADADIO User Manual*.

Macro	Description
ADADIO_GSC_AIDR	Analog Input Data Register
ADADIO_GSC_AOC0R	Analog Output Channel 0 Register
ADADIO_GSC_AOC1R	Analog Output Channel 1 Register
ADADIO_GSC_AOC2R	Analog Output Channel 2 Register
ADADIO_GSC_AOC3R	Analog Output Channel 3 Register
ADADIO_GSC_BCR	Board Control Register
ADADIO_GSC_BRR	Board Revision Register *
ADADIO_GSC_DIOPR	Digital I/O Port Register
ADADIO_GSC_SRR	Sample Rate Register

* The first time this register is read after a fresh load of the driver may take several seconds as access to this register requires an Auto-Calibration cycle and an initialization.

4.4.2.2. PCI Configuration Registers

Access to the PCI registers is seldom required so these registers are not listed here. For the complete list of PCI register identifiers refer to the driver's `gsc_pci9056.h` and `gsc_pci9080.h` header files, which are automatically included via `adadio.h`.

4.4.2.3. PLX PCI9056 and PCI9080 Feature Set Registers

Access to the PLX registers is seldom required so these registers are not listed here. For the complete list of PLX register identifiers refer to the driver's `gsc_pci9056.h` and `gsc_pci9080.h` header files, which are automatically included via `adadio.h`.

4.5. Data Types

The data types used by the API Library are described with the IOCTL services with which they are used.

4.6. Functions

The interface includes the following functions. The return values reflect the completion status of the requested operation. A value of zero indicates success. A negative value indicates that the request could not be completed successfully. The specific value returned is the negative of the corresponding error status value taken from `errno.h`. I/O services return positive values to indicate the number of bytes successfully transferred.

4.6.1. adadio_close()

This function is the entry point to close a connection to an open ADADIO board. The board is put in an initialized state before this call returns.

Prototype

```
int adadio_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. This is the negative of errno from errno.h.

Example

```
#include <stdio.h>

#include "adadio_dsl.h"

int adadio_close_dsl(int fd)
{
    int errs;
    int ret;

    ret = adadio_close(fd);

    if (ret)
        printf("ERROR: adadio_close() returned %d\n", ret);

    errs    = ret ? 1 : 0;
    return(errs);
}
```

4.6.2. adadio_init()

This function is the entry point to initializing the ADADIO 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 API calls return a failure status when the API Library is not initialized.

Prototype

```
int adadio_init(void);
```

Return Value	Description
0	The operation succeeded.
< 0	An error occurred. This is the negative of errno from errno.h.

Example

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

```
#include "adadio_dsl.h"

int adadio_init_dsl(void)
{
    int errs;
    int ret;

    ret = adadio_init();

    if (ret)
        printf("ERROR: adadio_init() returned %d\n", ret);

    errs    = ret ? 1 : 0;
    return(errs);
}
```

4.6.3. adadio_ioctl()

This function is the entry point to performing setup and control operations on an ADADIO board. This function should only be called after a successful open of the respective device. The specific operation performed varies according to the `request` argument. The `request` argument also governs the use and interpretation of the `arg` argument. The set of supported options for the `request` argument consists of the IOCTL services described below (section 4.7, page 23).

Prototype

```
int adadio_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 operation to be performed.
`arg`	This is a `request` specific argument. Refer to the IOCTL services for additional information (section 4.7, page 23).

Return Value	Description
`0`	The operation succeeded.
`< 0`	An error occurred. This is the negative of `errno` from `errno.h`.

Example

```
#include <stdio.h>

#include "adadio_dsl.h"

int adadio_ioctl_dsl(int fd, int request, void *arg)
{
    int errs;
    int ret;

    ret = adadio_ioctl(fd, request, arg);

    if (ret)
        printf("ERROR: adadio_ioctl() returned %d\n", ret);
```

```
    errs    = ret ? 1 : 0;
    return(errs);
}
```

4.6.4. adadio_open()

This function is the entry point to open a connection to an ADADIO board. The device is initialized before the function returns.

Prototype

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

Argument	Description
device	This is the zero-based index of the ADADIO to access. *
share	Open the device in Shared Access Mode? If non-zero the device is opened in Shared Access Mode (see below). If zero the device is opened in Exclusive Access Mode (see below).
fd	The device handle is returned here. The pointer cannot be NULL. Values returned are as follows.

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

* If the index value is -1, then the API Library accesses /proc/adadio.

Return Value	Description
0	The operation succeeded.
< 0	An error occurred. This is the negative of errno from errno.h.

Example

```
#include <stdio.h>

#include "adadio_dsl.h"

int adadio_open_dsl(int device, int share, int* fd)
{
    int errs;
    int ret;

    ret = adadio_open(device, share, fd);

    if (ret)
        printf("ERROR: adadio_open() returned %d\n", ret);

    errs    = ret ? 1 : 0;
    return(errs);
}
```

4.6.4.1. Access Modes

Shared Access Mode:

Shared Access Mode allows multiple applications to access the same device simultaneously. In this mode, the first successful open request returns with the device in an initialized state. Subsequent successful Shared Access Mode open requests do not affect the state of the device. Once opened in Shared Access Mode, the device access remains in this mode until all Shared Access Mode accesses release the device with a close request.

Exclusive Access Mode:

Exclusive Access Mode allows a single application to acquire exclusive access to a device. In this mode, a successful open request returns with the device in an initialized state. While open in this mode all subsequent open requests will fail regardless of the access mode requested. Once opened in Exclusive Access Mode, the device access remains in this mode until released by the application with a close request.

4.6.5. adadio_read()

This function is the entry point to reading data from an open ADADIO. This function should only be called after a successful open of the respective device. The function reads up to `bytes` bytes from the board. The return value is the number of bytes actually read.

> **NOTE:** When performing an open on device index `-1`, the API Library accesses the `/proc/adadio` text file. This read service then reads from that file. Refer to section 2.2, page 11.

> **NOTE:** For additional information please refer to the I/O Modes information (section 8.3, page 44).

Prototype

```
int adadio_read(int fd, void *dst, size_t bytes);
```

Argument	Description
`fd`	This is the file descriptor of the device to access.
`dst`	The data read will be put here.
`bytes`	This is the desired number of bytes to read. This must be a multiple of four (4).

Return Value	Description
`0` to `bytes`	The operation succeeded. A value less than `bytes` indicates that the request timed out.
`< 0`	An error occurred. This is the negative of `errno` from `errno.h`.

Example

```
#include <stdio.h>

#include "adadio_dsl.h"

int adadio_read_dsl(int fd, void* dst, size_t bytes, size_t* qty)
{
    int errs;
    int ret;

    ret = adadio_read(fd, dst, bytes);

    if (ret < 0)
        printf("ERROR: adadio_read() returned %d\n", ret);
```

```
    if (qty)
        qty[0]  = (ret < 0) ? 0 : (size_t) ret;

    errs    = (ret < 0) ? 1 : 0;
    return(errs);
}
```

4.6.6. ADADIO Write

A write function is not supported for generating analog output. Instead, applications must use the `ADADIO_IOCTL_AOUT_CH_X_WRITE` IOCTL services (section 4.7.9, page 26).

4.7. IOCTL Services

The ADADIO API Library and device driver implement the following IOCTL services. Each service is described along with the applicable `adadio_ioctl()` function arguments.

4.7.1. ADADIO_IOCTL_AIN_BUF_CLEAR

This service clears the data from the input buffer.

Usage

Argument	Description
`request`	`ADADIO_IOCTL_AIN_BUF_CLEAR`
`arg`	Not used.

4.7.2. ADADIO_IOCTL_AIN_BUF_ENABLE

This service enables and disables the input buffer.

Usage

Argument	Description
`request`	`ADADIO_IOCTL_AIN_BUF_ENABLE`
`arg`	`s32*`

Valid argument values are as follows.

Value	Description
`-1`	This requests the current setting.
`ADADIO_AIN_BUF_ENABLE_NO`	This disables input to the buffer and clears the buffer content.
`ADADIO_AIN_BUF_ENABLE_YES`	This enables input to the buffer.

4.7.3. ADADIO_IOCTL_AIN_BUF_SIZE

This service sets the size of the board's virtual input buffer. The physical buffer size is 32K samples deep.

NOTE: The buffer fill level status flags refer to the virtual buffer size, not the physical buffer size.

NOTE: Input sample collection is halted while the virtual buffer is full.

Usage

Argument	Description
request	ADADIO_IOCTL_AIN_BUF_SIZE
arg	s32*

Valid argument values are as follows.

Value	Description
-1	This requests the current setting.
ADADIO_AIN_BUF_SIZE_1	Set the Virtual Buffer size to one sample.
ADADIO_AIN_BUF_SIZE_2	Set the Virtual Buffer size to two samples.
ADADIO_AIN_BUF_SIZE_4	Set the Virtual Buffer size to four samples.
ADADIO_AIN_BUF_SIZE_8	Set the Virtual Buffer size to eight samples.
ADADIO_AIN_BUF_SIZE_16	Set the Virtual Buffer size to 16 samples.
ADADIO_AIN_BUF_SIZE_32	Set the Virtual Buffer size to 32 samples.
ADADIO_AIN_BUF_SIZE_64	Set the Virtual Buffer size to 64 samples.
ADADIO_AIN_BUF_SIZE_128	Set the Virtual Buffer size to 128 samples.
ADADIO_AIN_BUF_SIZE_256	Set the Virtual Buffer size to 256 samples.
ADADIO_AIN_BUF_SIZE_512	Set the Virtual Buffer size to 512 samples.
ADADIO_AIN_BUF_SIZE_1024	Set the Virtual Buffer size to 1,024 samples.
ADADIO_AIN_BUF_SIZE_2048	Set the Virtual Buffer size to 2,048 samples.
ADADIO_AIN_BUF_SIZE_4096	Set the Virtual Buffer size to 4,096 samples.
ADADIO_AIN_BUF_SIZE_8192	Set the Virtual Buffer size to 8,192 samples.
ADADIO_AIN_BUF_SIZE_16384	Set the Virtual Buffer size to 16,384 samples.
ADADIO_AIN_BUF_SIZE_32768	Set the Virtual Buffer size to 32,768 samples, which is the buffer's physical size.

4.7.4. ADADIO_IOCTL_AIN_BUF_STS

This service retrieves the fill level status of the virtual input buffer.

Usage

Argument	Description
request	ADADIO_IOCTL_AIN_BUF_STS
arg	s32*

Valid argument values returned are as follows.

Value	Description
ADADIO_AIN_BUF_STS_EMPTY	The virtual buffer is empty.
ADADIO_AIN_BUF_STS_ALMOST_EMPTY	The buffer is less than half full, but it is not empty.
ADADIO_AIN_BUF_STS_HALF_FULL	The buffer is at least half full, but it is not full.
ADADIO_AIN_BUF_STS_FULL	The virtual buffer is full.

4.7.5. ADADIO_IOCTL_AIN_CHAN_LAST

This service configures the selection of the last channel to scan, which effectively sets the range and number of channels to scan.

Usage

Argument	Description
request	ADADIO_IOCTL_AIN_CHAN_LAST
arg	s32*

Valid argument values are as follows.

Value	Description
-1	Retrieve the current setting.
0	Scan channel 0 only.
1	Scan channels 0-1.
2	Scan channels 0-2.
3	Scan channels 0-3.
4	Scan channels 0-4.
5	Scan channels 0-5.
6	Scan channels 0-6.
7	Scan channels 0-7.

4.7.6. ADADIO_IOCTL_AIN_MODE

This service configures the analog input mode.

Usage

Argument	Description
request	ADADIO_IOCTL_AIN_MODE
arg	s32*

Valid argument values are as follows.

Value	Description
-1	Passing in this value requests the current setting.
ADADIO_AIN_MODE_DIFF_BURST	This refers to Differential, burst input operation.
ADADIO_AIN_MODE_DIFF_CONT	This refers to Differential, continuous input operation.
ADADIO_AIN_MODE_LB_TEST	This refers to connection of a single output channel to all input channels.
ADADIO_AIN_MODE_SE_BURST	This refers to Single Ended, burst input operation.
ADADIO_AIN_MODE_SE_CONT	This refers to Single Ended, continuous input operation.
ADADIO_AIN_MODE_VREF_TEST	This option connects the inputs to the +VREF reference voltage.
ADADIO_AIN_MODE_ZERO_TEST	This option connects the inputs to the zero-reference voltage.

4.7.7. ADADIO_IOCTL_AIN_NRATE

This service sets the NRATE divider value for input sample rate generation.

Usage

Argument	Description
request	ADADIO_IOCTL_AIN_NRATE
arg	s32*

Valid argument values are as follows.

Value	Description
-1	Passing in this value requests the current setting.
100 – 0xFFFF	This is the range for 200K S/S boards
200 – 0xFFFF	This is the range for 100K S/S boards

4.7.8. ADADIO_IOCTL_AIN_TRIGGER

This service initiates an input strobe operation.

Usage

Argument	Description
request	ADADIO_IOCTL_AIN_TRIGGER
arg	Not used.

4.7.9. ADADIO_IOCTL_AOUT_CH_*X*_WRITE

This refers to the below listed services.

Service	Description
ADADIO_IOCTL_AOUT_CH_0_WRITE	Write to Output Channel 0.
ADADIO_IOCTL_AOUT_CH_1_WRITE	Write to Output Channel 1.
ADADIO_IOCTL_AOUT_CH_2_WRITE	Write to Output Channel 2.
ADADIO_IOCTL_AOUT_CH_3_WRITE	Write to Output Channel 3.

These services write a value to their respective analog output channels.

Usage

Argument	Description
request	ADADIO_IOCTL_AOUT_CH_*X*_WRITE
arg	s32*

Valid argument values are as follows.

Value	Description
-1	Passing in this value requests the current setting.
0x0000 – 0xFFFF	This is the value to be applied to the output channel.

4.7.10. ADADIO_IOCTL_AOUT_ENABLE

This service enables or disables the analog outputs.

Usage

Argument	Description
request	ADADIO_IOCTL_AOUT_ENABLE
arg	s32*

Valid argument values are as follows.

Value	Description
-1	Passing in this value requests the current setting.
ADADIO_AOUT_ENABLE_NO	Disable the outputs.

ADADIO_AOUT_ENABLE_YES	Enable the outputs

4.7.11. ADADIO_IOCTL_AOUT_STROBE

This service initiates an output strobe operation.

Usage

Argument	Description
request	ADADIO_IOCTL_AOUT_STROBE
arg	Not used.

4.7.12. ADADIO_IOCTL_AOUT_STROBE_ENABLE

This service enables or disables output strobe operation.

Usage

Argument	Description
request	ADADIO_IOCTL_AOUT_STROBE_ENABLE
arg	s32*

Valid argument values are as follows.

Value	Description
-1	Passing in this value requests the current setting.
ADADIO_AOUT_STROBE_ENABLE_NO	Disable output strobe operation. Values are posted to the output immediately.
ADADIO_AOUT_STROBE_ENABLE_YES	Enable output strobe operation. Values are posted to the output only in response to an output strobe.

4.7.13. ADADIO_IOCTL_AUTO_CALIBRATE

This service initiates an Auto-Calibration cycle. The service returns when the operation completes, which is on the order of about 10 seconds.

> **NOTE:** The driver performs an initialization after performing an auto-calibration, as required by the hardware. This is a complete initialization and is the same as is done with the ADADIO_IOCTL_INITIALIZE IOCTL service (section 4.7.20, page 29).

> **NOTE:** If the auto-calibration service returns an error status, an error message will be posted to the system log briefly describing the error condition.

Usage

Argument	Description
request	ADADIO_IOCTL_AUTO_CALIBRATE
arg	Not used.

4.7.14. ADADIO_IOCTL_DATA_FORMAT

This service sets the analog input and output data encoding format.

Usage

Argument	Description
request	ADADIO_IOCTL_DATA_FORMAT
arg	s32*

Valid argument values returned are as follows.

Value	Description
-1	This requests the current setting.
ADADIO_DATA_FORMAT_2S_COMP	This refers to the Twos Compliment encoding format.
ADADIO_DATA_FORMAT_OFF_BIN	This refers to the Offset Binary encoding format.

4.7.15. ADADIO_IOCTL_DIO_PIN_READ

This service reads the dedicated digital input pin.

Usage

Argument	Description
request	ADADIO_IOCTL_DIO_PIN_READ
arg	s32*

Valid argument values returned are as follows.

Value	Description
ADADIO_DIO_PIN_CLEAR	The dedicated input is low.
ADADIO_DIO_PIN_SET	The dedicated input is high.

4.7.16. ADADIO_IOCTL_DIO_PIN_WRITE

This service sets the output level for the dedicated digital output pin.

Usage

Argument	Description
request	ADADIO_IOCTL_DIO_PIN_WRITE
arg	s32*

Valid argument values are as follows.

Value	Description
-1	This requests the current setting.
ADADIO_DIO_PIN_CLEAR	Set the output to a low level.
ADADIO_DIO_PIN_SET	Set the output to a high level.

4.7.17. ADADIO_IOCTL_DIO_PORT_DIR

This service sets the direction of the 8-bit digital I/O port.

Usage

Argument	Description
request	ADADIO_IOCTL_DIO_PORT_DIR

arg	s32*

Valid argument values are as follows.

Value	Description
-1	This requests the current setting.
ADADIO_DIO_PORT_DIR_INPUT	This selects the input direction.
ADADIO_DIO_PORT_DIR_OUTPUT	This selects the output direction.

4.7.18. ADADIO_IOCTL_DIO_PORT_READ

This service reads the value at the digital I/O port. The value read is input data only if the port is configured as an input. If the port is configured to output data, then this service retrieves the current output value.

Usage

Argument	Description
request	ADADIO_IOCTL_DIO_PORT_READ
arg	s32*

Valid argument values returned are as follows.

Value	Description
0x00 – 0xFF	This is an 8-bit port.

4.7.19. ADADIO_IOCTL_DIO_PORT_WRITE

This service sets the output value for the digital I/O port. The value provided appears at the digital I/O port only if the port is configured as an output.

Usage

Argument	Description
request	ADADIO_IOCTL_DIO_PORT_WRITE
arg	s32*

Valid argument values are as follows.

Value	Description
-1	Passing in this value returns the last output value, if the port is configured as an output. If the port is configured as an input, then this will retrieve the current input.
0x00 – 0xFF	This is an 8-bit port.

4.7.20. ADADIO_IOCTL_INITIALIZE

This service returns all interface settings for the board to the state they were in when the board was first opened. This includes both hardware-based settings and software-based settings.

> **NOTE:** If the initialization service returns an error status, an error message will be posted to the system log briefly describing the error condition.

Usage

Argument	Description
request	ADADIO_IOCTL_INITIALIZE
arg	Not used.

4.7.21. ADADIO_IOCTL_IRQ_SEL

This service configures the source selection for firmware interrupts.

Usage

Argument	Description
request	ADADIO_IOCTL_IRQ_SEL
arg	s32*

Valid argument values are as follows.

Value	Description
-1	Retrieve the current setting.
ADADIO_IRQ_AUTO_CAL_DONE	This refers to the completion of an Auto-Calibration cycle.
ADADIO_IRQ_AIN_BUF_EMPTY	This refers to the Input Buffer becoming empty.
ADADIO_IRQ_AIN_BUF_FULL	This refers to the Input Buffer becoming full.
ADADIO_IRQ_AIN_BUF_HALF_FULL	This refers to the Input Buffer becoming half full.
ADADIO_IRQ_AIN_BURST_DONE	This refers to the completion of an input burst.
ADADIO_IRQ_AOUT_STROBE_DONE	This refers to the completion of an output strobe.
ADADIO_IRQ_INIT_DONE	This refers to the completion of an initialization cycle.

4.7.22. ADADIO_IOCTL_LOOPBACK_CHANNEL

This service selects the output channel to use for the Loopback input mode selection.

Usage

Argument	Description
request	ADADIO_IOCTL_LOOPBACK_CHANNEL
arg	s32*

Valid argument values are as follows.

Value	Description
-1	This requests the current setting.
0	Select channel 0 output as the input source.
1	Select channel 1 output as the input source.
2	Select channel 2 output as the input source.
3	Select channel 3 output as the input source.

4.7.23. ADADIO_IOCTL_QUERY

This service queries the driver for various pieces of information about the board and the driver.

Usage

Argument	Description
request	ADADIO_IOCTL_QUERY
arg	s32*

Valid argument values are as follows.

Value	Description
ADADIO_QUERY_AUTO_CAL_MS	This returns the maximum duration of the Auto Calibration cycle in milliseconds.
ADADIO_QUERY_COUNT	This returns the number of query options supported by the IOCTL service.
ADADIO_QUERY_DEVICE_TYPE	This returns the identifier value for the board's type. This should be GSC_DEV_TYPE_ADADIO.
ADADIO_QUERY_DMDMA	Does the board support Demand Mode DMA? (0 = no, 1 = yes)
ADADIO_QUERY_FIFO_SIZE	This returns the size of the input buffer in 32-bit A/D values.
ADADIO_QUERY_FSAMP_MAX	This gives the maximum FSAMP value in S/S.
ADADIO_QUERY_FSAMP_MIN	This gives the minimum FSAMP value in S/S.
ADADIO_QUERY_INIT_MS	This returns the duration of a board initialization in milliseconds.
ADADIO_QUERY_MASTER_CLOCK	This returns the master clock frequency in hertz.
ADADIO_QUERY_NRATE_MASK	This returns the mask for the board's NRATE fields.
ADADIO_QUERY_NRATE_MAX	This returns the maximum supported NRATE value.
ADADIO_QUERY_NRATE_MIN	This returns the minimum supported NRATE value.
ADADIO_QUERY_VRANGE	This returns the voltage range option supported by the board. See below.

Valid return values are as indicated in the above table and as given in the below table.

Value	Description
ADADIO_IOCTL_QUERY_ERROR	Either there was a processing error or the query option is unrecognized.

Valid return values for the ADADIO_QUERY_VRANGE option are as follows.

Value	Description
ADADIO_VRANGE_10	The board is configured for ±10V.
ADADIO_VRANGE_25_5_10	The board is configured for ±2.5V, ±5V or ±10V.

4.7.24. ADADIO_IOCTL_REG_MOD

This service performs a read-modify-write operation on a board register. This includes only the firmware registers, as the PCI and PLX Feature Set registers are read-only.

Usage

Argument	Description
request	ADADIO_IOCTL_REG_MOD
arg	gsc_reg_t*

Definition

```
typedef struct
{
```

```
    u32 reg;    // range: any valid register definition
    u32 value;  // range: 0x0-0xFFFFFFFF
    u32 mask;   // range: 0x0-0xFFFFFFFF
} gsc_reg_t;
```

Fields	Description
reg	This is the register to access. Refer to section 4.4.2 on page 18 for additional information.
value	This is the value to write to the specified register. Only the bits set in the map are applied.
mask	This is a map of the bits to modify. If a bit is set, then the corresponding register bit is set according the content of the value field. If a bit here is zero, then that register bit is unmodified.

4.7.25. ADADIO_IOCTL_REG_READ

This service reads the value of an ADADIO register. This includes the PCI registers, the PLX Feature Set registers and the firmware registers.

Usage

Argument	Description
request	ADADIO_IOCTL_REG_READ
arg	gsc_reg_t*

Definition

```
typedef struct
{
    u32 reg;    // range: any valid register definition
    u32 value;  // range: 0x0-0xFFFFFFFF
    u32 mask;   // range: 0x0-0xFFFFFFFF
} gsc_reg_t;
```

Fields	Description
reg	This is the register to read from. Refer to section 4.4.2 on page 18 for additional information.
value	This is the value read from the specified register.
mask	This is ignored for read requests.

4.7.26. ADADIO_IOCTL_REG_WRITE

This service writes a value to a board register. This includes only the firmware registers, as the PCI and PLX Feature Set registers cannot be modified.

Usage

Argument	Description
request	ADADIO_IOCTL_REG_WRITE
arg	gsc_reg_t*

Definition

```
typedef struct
{
    u32 reg;    // range: any valid register definition
    u32 value;  // range: 0x0-0xFFFFFFFF
    u32 mask;   // range: 0x0-0xFFFFFFFF
```

```
} gsc_reg_t;
```

Fields	Description
reg	This is the register to write to. Refer to section 4.4.2 on page 18 for additional information.
value	This is the value to write to the specified register.
mask	This is ignored for write requests.

4.7.27. ADADIO_IOCTL_RX_IO_ABORT

This service aborts an ongoing `adadio_read()` request.

Usage

Argument	Description
request	ADADIO_IOCTL_RX_IO_ABORT
arg	s32*

The results are reported as one of the following values.

Value	Description
ADADIO_IO_ABORT_NO	A read request was not aborted as none were ongoing.
ADADIO_IO_ABORT_YES	An ongoing read request was aborted.

4.7.28. ADADIO_IOCTL_RX_IO_MODE

This service selects the data transfer mode for I/O operations. Refer to the `adadio_read()` service for additional information (section 4.6.5 on page 22).

Usage

Argument	Description
request	ADADIO_IOCTL_RX_IO_MODE
arg	s32*

Valid argument values are as follows.

Value	Description
-1	This requests the current setting.
GSC_IO_MODE_BMDMA	This refers to Block Mode DMA in which the DMA is initiated only after the data becomes available.
GSC_IO_MODE_DMDMA	This refers to Demand Mode DMA in which the transfer occurs as the data become available. This is the most efficient option for most I/O requests.
GSC_IO_MODE_PIO	This refers to PIO in which data is transferred by repetitive register accesses. This is preferred for very small transfer requests. This is the default.

NOTE: Demand Mode DMA is not available on boards with older firmware. Refer to the board hardware manual for details, or to the ADADIO_QUERY_DMDMA query option (section 4.7.23, page 30).

4.7.29. ADADIO_IOCTL_RX_IO_TIMEOUT

This service sets the timeout limit for I/O requests. The limit is specified in seconds.

Usage

Argument	Description
request	ADADIO_IOCTL_RX_IO_TIMEOUT
arg	s32*

Valid argument values are in the range from zero to 3600, -1, and ADADIO_IO_TIMEOUT_INFINITE. A value of zero tells the driver not to sleep in order to wait for more data, and should only be used with PIO mode reads. A value of -1 is used to retrieve the current setting. If the option ADADIO_IO_TIMEOUT_INFINITE is used, then the driver will wait indefinitely rather than timing out. The default is 10 seconds.

4.7.30. ADADIO_IOCTL_WAIT_CANCEL

This service resumes all threads blocked via ADADIO_IOCTL_WAIT_EVENT IOCTL calls (section 4.7.31, page 35), according to the provided criteria. When a blocked thread is waiting for any event specified in the structure, then the thread is resumed.

> **NOTE:** The driver itself makes use of the wait services for various internal operations. Driver initiated waits are unaffected by application cancel requests.

Usage

Argument	Description
request	ADADIO_IOCTL_WAIT_CANCEL
arg	gsc_wait_t*

Definition

```
typedef struct
{
    u32  flags;
    u32  main;
    u32  gsc;
    u32  alt;
    u32  io;
    u32  timeout_ms;
    u32  count;
} gsc_wait_t;
```

Fields	Description
flags	This is unused by wait cancel operations.
main	This specifies the set of GSC_WAIT_MAIN_* events whose wait requests are to be cancelled. Refer to section 4.7.31.2 on page 36.
gsc	This specifies the set of ADADIO_WAIT_GSC_* events whose wait requests are to be cancelled. Refer to section 4.7.31.3 on page 36.
alt	This is unused with the ADADIO board and should be zero.
io	This specifies the set of ADADIO_WAIT_IO_* events whose wait requests are to be cancelled. Refer to section 4.7.31.4 on page 36.
timeout_ms	This is unused by wait cancel operations.
count	Upon return this indicates the number of waits that were cancelled.

4.7.31. ADADIO_IOCTL_WAIT_EVENT

This service blocks a thread until any one of a specified set of events occurs, or until a timeout lapses, whichever occurs first. The set of possible events to wait for are specified in the structure's `main`, `gsc`, `alt` and `io` fields. All field values must be valid and at least one event must be specified. If the thread is resumed because one of the referenced events has occurred, then the bit for the respective event is the only event bit that will be set. All other event bits and fields will be zero. (Multiple event bits will be set only if the events occur simultaneously.)

> **NOTE**: The service waits only for the first of the specified events, not for all specified events.
>
> **NOTE:** A wait timeout is reported via the `gsc_wait_t` structure's `flags` field having the `GSC_WAIT_FLAG_TIMEOUT` flag set, rather than via an `ETIMEDOUT` error.

Usage

Argument	Description
request	ADADIO_IOCTL_WAIT_EVENT
arg	gsc_wait_t*

Definition

```
typedef struct
{
    u32  flags;
    u32  main;
    u32  gsc;
    u32  alt;
    u32  io;
    u32  timeout_ms;
    u32  count;
} gsc_wait_t;
```

Fields	Description
flags	This must initially be zero. Upon return this indicates the reason that the thread was resumed. Refer to section 4.7.31.1on page 35.
main	This specifies any number of `GSC_WAIT_MAIN_*` events that the thread is to wait for. Refer to section 4.7.31.2 on page 36.
gsc	This specifies any number of `ADADIO_WAIT_GSC_*` events that the thread is to wait for. Refer to section 4.7.31.3 on page 36.
alt	This is unused with the ADADIO board and must be zero.
io	This specifies any number of `ADADIO_WAIT_IO_*` events that the thread is to wait for. Refer to section 4.7.31.4 on page 36.
timeout_ms	This specified the maximum amount of time, in milliseconds, that the thread is to wait for any of the referenced events. A value of zero means do not timeout at all. If non-zero, then upon return the value will be the approximate amount of time actually waited.
count	This is unused by wait event operations and must be zero.

4.7.31.1. `gsc_wait_t.flags` Options

Upon return from a wait request the wait structure's `flags` field will indicate the reason that the thread was resumed. Only one of the below options will be set.

Fields	Description
GSC_WAIT_FLAG_CANCEL	The wait request was cancelled.

GSC_WAIT_FLAG_DONE	One of the referenced events occurred.
GSC_WAIT_FLAG_TIMEOUT	The timeout period lapsed before a referenced event occurred.

4.7.31.2. gsc_wait_t.main Options

The wait structure's main field may specify any of the below primary interrupt options. These interrupt options are supported by the ADADIO and other General Standards products.

Fields	Description
GSC_WAIT_MAIN_DMA0	This refers to the DMA Done interrupt on DMA engine number zero.
GSC_WAIT_MAIN_DMA1	This refers to the DMA Done interrupt on DMA engine number one.
GSC_WAIT_MAIN_GSC	This refers to any of the Interrupt Control/Status Register interrupts.
GSC_WAIT_MAIN_OTHER	This generally refers to an interrupt generated by another device sharing the same interrupt as the ADADIO.
GSC_WAIT_MAIN_PCI	This refers to any interrupt generated by the ADADIO.
GSC_WAIT_MAIN_SPURIOUS	This refers to board interrupts which should never be generated.
GSC_WAIT_MAIN_UNKNOWN	This refers to board interrupts whose source could not be identified.

4.7.31.3. gsc_wait_t.gsc Options

The wait structure's gsc field may specify any combination of the below interrupt options. These are the interrupt options referenced in the Board Control Register. Applications are responsible for selecting the desired interrupts. Refer to ADADIO_IOCTL_IRQ_SEL (section 4.7.21, page 30).

Value	Description
ADADIO_WAIT_GSC_AIN_BUF_EMPTY	This refers to the Input Buffer becoming empty.
ADADIO_WAIT_GSC_AIN_BUF_FULL	This refers to the Input Buffer becoming full.
ADADIO_WAIT_GSC_AIN_BUF_HALF_FULL	This refers to the Input Buffer becoming half full.
ADADIO_WAIT_GSC_AIN_BURST_DONE	This refers to the completion of an input burst.
ADADIO_WAIT_GSC_AOUT_STROBE_DONE	This refers to the completion of an output strobe.
ADADIO_WAIT_GSC_AUTO_CAL_DONE	This refers to the completion of an Auto-Calibration cycle.
ADADIO_WAIT_GSC_INIT_DONE	This refers to the completion of an initialization cycle.

4.7.31.4. gsc_wait_t.io Options

The wait structure's io field may specify any of the below event options. These events are generated in response to application board data read requests.

Fields	Description
ADADIO_WAIT_IO_RX_ABORT	This refers to read requests which have been aborted.
ADADIO_WAIT_IO_RX_DONE	This refers to read requests which have been satisfied.
ADADIO_WAIT_IO_RX_ERROR	This refers to read requests which end due to an error.
ADADIO_WAIT_IO_RX_TIMEOUT	This refers to read requests which end due to the timeout period lapse.

4.7.32. ADADIO_IOCTL_WAIT_STATUS

This service count all threads blocked via the ADADIO_IOCTL_WAIT_EVENT IOCTL service (section 4.7.31, page 35), according to the provided criteria. A match is made when a waiting thread's wait criteria matches any of the criteria specified in the structure passed to this service.

> **NOTE:** The driver itself makes use of the wait services for various internal operations. Driver initiated waits are ignored by application status requests.

Usage

Argument	Description
request	ADADIO_IOCTL_WAIT_STATUS
arg	gsc_wait_t*

Definition

```
typedef struct
{
    u32  flags;
    u32  main;
    u32  gsc;
    u32  alt;
    u32  io;
    u32  timeout_ms;
    u32  count;
} gsc_wait_t;
```

Fields	Description
flags	This is unused by wait status operations.
main	This specifies the set of GSC_WAIT_MAIN_* events whose wait requests are to be counted. Refer to section 4.7.31.2 on page 36.
gsc	This specifies the set of ADADIO_WAIT_GSC_* events whose wait requests are to be counted. Refer to section 4.7.31.3 on page 36.
alt	This is unused with the ADADIO board and should be zero.
io	This specifies the set of ADADIO_WAIT_IO_* events whose wait requests are to be counted. Refer to section 4.7.31.4 on page 36.
timeout_ms	This is unused by wait status operations.
count	Upon return this indicates the number of waits that met any of the specified criteria.

5. The Driver

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

5.1. Files

The device driver files are summarized in the table below.

File	Description
`driver/*.c`	The driver source files.
`driver/*.h`	The driver header files.
`driver/adadio.h`	This is the driver interface header file.
`driver/Makefile`	This is the driver make file.
`driver/start`	Shell script to install the driver executable and device nodes.
`driver/adadio.ko`	This is the driver executable (kernel 2.6 and later).
`driver/adadio.o`	This is the driver executable (kernel 2.4 and earlier).

5.2. Build

NOTE: Building the driver requires installation of the kernel sources.

The device driver is built via the Overall Make Script (section 2.7, page 12), but can be built separately following the below steps.

1. Change to the directory where the driver and its sources are installed (…`/driver/`).

2. Remove existing build targets using the below command.

   ```
   make clean
   ```

3. Build the driver by issuing the below command.

   ```
   make
   ```

   NOTE: Due to the differences between the many Linux distributions some build errors may occur. These errors may include system header location differences, which should be easily corrected.

5.3. Startup

NOTE: The driver will have to be built before being used as it is provided in source form only.

The startup script used in this procedure is designed to ensure that the driver module in the install directory is the module that is loaded. This is accomplished by making sure that an already loaded module is first unloaded before attempting to load the module from the disk drive. In addition, the script also deletes and recreates the device nodes. This is done to ensure that the device nodes in use have the same major number as assigned dynamically to the driver by the kernel, and so that the number of device nodes correspond to the number of boards identified by the driver.

5.3.1. Manual Driver Startup Procedures

Start the driver manually by following the below listed steps.

NOTE: The following steps may require elevated privileges.

1. Change to the directory where the driver sources are installed (…`/driver/`).

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

   ```
   ./start
   ```

   NOTE: This script must be executed each time the host is rebooted.

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

3. Verify that the device driver module has been loaded by issuing the below command and examining the output. The module name `adadio` should be included in the output.

   ```
   lsmod
   ```

4. Verify that the device nodes have been created by issuing the below command and examining the output. The output should include one node for each installed board.

   ```
   ls –l /dev/adadio.*
   ```

5.3.2. Automatic Driver Startup Procedures

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

1. Locate and edit the system startup script `rc.local`, which should be in the `/etc/rc.d/` directory. Modify the file by adding the below line so that it is executed with every reboot. The example is based on the driver being installed in `/usr/src/linux/drivers/`, though it may have been installed elsewhere.

   ```
   /usr/src/linux/drivers/adadio/driver/start
   ```

   NOTE: For `systemd` installations the file `rc.local` may be located under the `/etc/` directory rather than under `/etc/rc.d/`.

2. Load the driver and create the required device nodes by rebooting the system.

3. Verify that the driver is loaded and that the device nodes have been created. Do this by following the verification steps given in the manual startup procedures.

5.3.2.1. File `rc.local` Not Present

Some distributions may not install a default version of `rc.local`. Some may not even create the directory `/etc/rc.d/`. If the directory is not present, then it may be created. The directory must be created with the owner and group set to `root`. The directory permissions must be set to `rwxr-xr-x`. If the file `/etc/rc.d/rc.local` is not present, then it too may be created. The file must also be created with the owner and group set to `root`. Additionally, the file permissions must also be set to `rwxr-xr-x`. After the directory and file are created as described, reboot to verify boot time loading of the driver. Here is an example of a default version of `rc.local`.

```
#!/bin/bash

# Add you local content here.
```

5.3.2.2. Default `rc.local` File Permissions

The `rc.local` script may fail to run at boot time because some distributions install a default version of the file without execute permissions. Without execute permissions, boot time invocation of the script fails, which inhibits boot time loading of the driver. If this is the case, then change the file permissions to `rwxr-xr-x`. After the file permissions are adjusted as described, reboot to verify boot time loading of the driver.

5.3.2.3. `systemd` Installations

With the advent of the `systemd` startup implementation, `rc.local` may be accessed via a `systemd` startup service. The service name may be `rc-local`, `rc-local.service` or something similar. This service may or may not be enabled by default. If the service is disabled, then the script will not execute, which prevents boot time loading of the driver. The service can be enabled with the below command line. After the service is enabled, reboot to verify boot time loading of the driver.

```
systemctl enable rc-local
```

NOTE: For `systemd` installations the file `rc.local` may be located under the `/etc/` directory rather than under `/etc/rc.d/`.

5.3.2.4. `systemd` and `rc.local` Timing

If the above steps have been performed but the driver still does not start then examine the `dmesg` output for driver messages. If the output shows that the driver starts and immediately stops, then the problem may be timing. That is, since `systemd` doesn't serialize startup initialization as done in the past, driver loading may fail if required services have not completed their own initialization. If this is the problem, then it may be corrected simply by inserting a delay in `rc.local` prior to it calling the driver's `start` script (i.e., `sleep` for one or more seconds).

5.3.2.5. SElinux Implications

If not disabled, then SElinux may prevent boot time loading of the driver. If this is the case, then it can be verified and corrected using SElinux related tools and utilities. First, install the necessary software using the below command. (As necessary, replace the `yum` command line with that which is available for your distribution.)

```
yum install setroubleshoot setools
```

Next, run the below command to determine if SElinux is preventing the driver from loading at boot time.

```
sealert –a /var/log/audit/audit.log
```

If SElinux is preventing the driver from loading, then the output from the above command should include a reference to the driver's `start` script, the `insmod` command that loads the driver or the name of the driver executable. If so, then the output should also indicate the commands necessary to resolve the issue. The following is an example of the instructions given when the culprit is `insmod`, which is the start script command that loads the driver. After running these commands reboot the system to verify boot time loading of the driver.

```
ausearch -c 'insmod' --raw | audit2allow -M my-insmod
semodule -X 300 -i my-insmod.pp
```

5.4. Verification

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

1. Verify that the file `/proc/adadio` is present. If the file is present then the driver is loaded and running. Verify the file's presence by viewing its content with the below command.

   ```
   cat /proc/adadio
   ```

5.5. Version

The driver version number can be obtained in a variety of ways. It is reported by the driver both when the driver is loaded and when it is unloaded (depending on kernel configuration options, this may be visible only in places such as `/var/log/messages`). It is reported in the text file `/proc/adadio` while the driver is loaded and running. The version number is also given in the file `release.txt` in the root install directory.

5.6. Shutdown

Shutdown the driver following the below listed steps.

> **NOTE**: The following steps may require elevated privileges.

1. If the driver is currently loaded then issue the below command to unload the driver.

   ```
   rmmod adadio
   ```

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

   ```
   lsmod
   ```

6. Document Source Code Examples

The source code examples included in this document are built into a statically linkable library usable with console applications. The purpose of these files is to verify that the documentation samples compile and to provide a library of working sample code to assist in a user's learning curve and application development effort.

6.1. Files

The library files are summarized in the table below.

File	Description
`docsrc/*.c`	These are the C source files.
`docsrc/makefile`	This is the library make file.
`docsrc/makefile.dep`	This is an automatically generated make dependency file.
`include/adadio_dsl.h`	This is the primary utility header file.
`lib/adadio_dsl.a`	This is the statically linkable library file.

6.2. Build

The library is built via the Overall Make Script (section 2.7, page 12), but can be built separately following the below steps.

1. Change to the directory where the documentation sources are installed (…`/docsrc/`).

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

   ```
   make clean
   ```

3. Compile the sample files and build the library by issuing the below command.

   ```
   make
   ```

4. Rebuild the Main Library (section 3.2, page 15).

6.3. Library Use

The library is used both at application compile time and at application link time. At compile time include the below listed header file in each source file using a component of the library interface. At link time include the below listed library file with the objects being linked with the application.

Description	File	Location
Header File	`adadio_dsl.h`	…`/include/`
Static Link Library	`adadio_dsl.a`	…`/lib/`

7. Utility Source Code

The driver archive includes a body of utility services built into a statically linkable library that is usable with console applications. The primary purpose of the services is both for code reuse in the sample applications and to provide wrappers, mostly visual, around the driver's IOCTL services. The aim of the visual wrappers is to facilitate structured console output for the sample applications. An additional purpose of these utility services is to provide a library of working sample code to assist in a user's learning curve and application development effort.

7.1. Files

The library files are summarized in the table below.

File	Description
utils/*.c	These are device specific utility source files.
utils/gsc_*.c	These are device and OS independent utility source files.
utils/os_*.c	These are OS specific utility source files.
utils/makefile	This is the library make file.
utils/makefile.dep	This is an automatically generated make dependency file.
include/adadio_utils.h	This is the primary utility header file.
lib/adadio_utils.a	This is the statically linkable library file.

7.2. Build

The library is built via the Overall Make Script (section 2.7, page 12), but can be built separately following the below steps.

1. Change to the directory where the utility sources are installed (…/utils/).

2. Remove existing build targets using the below command.

   ```
   make clean
   ```

3. Compile the sample files and build the libraries using the below command.

   ```
   make
   ```

4. Rebuild the Main Library (section 3.2, page 15).

7.3. Library Use

The library is used both at application compile time and at application link time. At compile time include the below listed header file in each source file using a component of the library interface. At link time include the below listed library files with the objects being linked with the application.

Description	File	Location
Header File	adadio_utils.h	…/include/
Static Link Libraries	adadio_utils.a gsc_utils.a os_utils.a plx_utils.a	…/lib/

8. Operating Information

This section explains some basic operational procedures for using the ADADIO. This is in no way intended to be a comprehensive guide. This is simply to address a very few issues relating to their use.

8.1. Debugging Aids

The driver package includes the following items useful for development and/or debugging aids.

8.1.1. Device Identification

When communicating with technical support complete device identification is virtually always necessary. The *id* example application is provided for this specific purpose. This is a text only console application. The output can be piped to a file, which can then be emailed to GSC technical support when requested. Locate the application as follows.

Description	File	Location
Application	`id`	`…/id/`

8.1.2. Detailed Register Dump

Among the utility services provided is a function to generate a detailed listing of the board's registers to the console. When used, the function is typically used to verify the board's configuration. In these cases, the function should be called just prior to the first read or write operation. When intended for sending to GSC tech support, please set the *detail* argument to `1`. The function arguments are as follows. The utility location is given in the subsequent table.

Argument	Description
`fd`	This is the file descriptor used to access the device.
`detail`	If non-zero the GSC register dump will include details of each register field.

Description	File/Name	Location
Function	`adadio_reg_list()`	Source File
Source File	`util_reg.c`	`…/utils/`
Header File	`adadio_utils.h`	`…/include/`
Library File	`adadio_utils.a`	`…/lib/`

8.2. Analog Input Configuration

The basic steps for Analog Input configuration are illustrated in the utility function noted below. The table also gives the location of the source file, the header file and the corresponding library containing the executable code.

Item	Name/File	Location
Function	`adadio_config_ai()`	Source File
Source File	`util_config_ai.c`	`…/utils/`
Header File	`adadio_utils.h`	`…/include/`
Library File	`adadio_utils.a`	`…/lib/`

8.3. I/O Modes

8.3.1. PIO - Programmed I/O

This mode involves repetitive register accesses. In this mode the driver will write data to the output buffer one value at a time. As needed, the driver will repeatedly sleep for one system time tick in order to wait for addition space in

the output buffer. This process is repeated until the data is exhausted or the I/O timeout expires, whichever occurs first.

8.3.2. BMDMA - Block Mode DMA

This mode is intended for data transfers that do not exceed the size of the ADADIO input buffer. Here, the board's DMA engine is used to perform a hardware-controlled transfer which does not require processer intervention to move the data. In this mode the DMA transfer is initiated only when the input buffer contains sufficient data to fulfill the request. This is a very efficient I/O method. However, for small requests PIO is more efficient.

8.3.3. DMDMA - Demand Mode DMA

This DMA transfer mode is similar to the block mode, except that a transfer for the entire amount of data is initiated immediately and is not limited to the size of the virtual FIFO. Here however, the actual movement of data occurs as the data becomes available in the input buffer. This is the most efficient method supported. However, for small requests PIO is more efficient.

9. Sample Applications

The driver archive includes a variety of sample and test applications. While they are provided without support and without any external documentation, any problems reported will be addressed as time permits. The applications are command line based and produce text output for display on a console. All of the applications are built via the Overall Make Script (section 2.7, page 12), but each may be built individually by changing to its respective directory and issuing the commands "`make clean`" and "`make all`". The initial output from each application includes information on its supported command line arguments. The following gives a brief overview of each application.

9.1. aout - Analog Output - …/aout/

This application outputs a repeating pattern on the four output channels. The pattern is different for each channel, though they are synchronized at the same modest rate.

9.2. din - Digital Input - …/din/

This application reads the cable's digital I/O signals and reports the values read to the console.

9.3. dout - Digital Output - …/dout/

This application writes a pattern to the cable's digital output lines as it is displayed to the console.

9.4. id - Identify Board - …/id/

This application reports detailed board identification information. This can be used with tech support to help identify as much technical information about the board as possible from software.

9.5. regs - Register Access - …/regs/

This application provides menu based interactive access to the board's registers, and reports other pertinent information to the console.

9.6. rxrate - Receive Rate - …/rxrate/

This application configures the board for its highest ADC sample rate then reads the input as fast as possible. The purpose is to measure the peak sustainable input rate for the host, per the provided command line arguments.

9.7. savedata - Save Acquired Data - …/savedata/

This application configures the board for a modest sample rate, reads a megabyte of data, then saves the data to a hex file.

9.8. sbtest - Single Board Test - …/sbtest/

This application performs functional testing of the driver and a user specified board, at least to the extent possible with just a single board and no additional equipment.

Document History

Revision	Description
January 30, 2023	Updated to version 4.9.102.44.0. Updated the kernel support table. Added section on environment variables. Updated the information for the open and close calls. Minor editorial modifications.
March 8, 2022	Updated to version 4.8.97.38.0. Updated the kernel support table. Expanded automatic startup information. Minor editorial changes.
January 28, 2020	Updated to version 4.7.90.30.1.
January 20, 2020	Updated to version 4.7.90.30.0. Minor editorial changes.
December 5, 2019	Updated to version 4.6.89.29.0. Minor editorial changes. Added a licensing subsection. Added WAIT_EVENT note. Added support for the ADADIO2.
June 25, 2019	Updated to version 4.5.86.28.0. Updated the kernel support table. Minor editorial changes. Some document reorganization.
February 7, 2019	Updated to version 4.4.81.26.0. Updated the inside cover page. Updated the CPU and kernel support section. Minor editorial changes. Updated Block Mode DMA macro and associated information. Document reorganization.
December 5, 2016	Updated to version 4.3.68.18.0. Updated the kernel support table. Added support for infinite I/O timeouts. Updated the operating information section. Made various miscellaneous updates. Some document reorganization.
June 9, 2016	Updated to version 4.2.66.14.0. Organized sample applications alphabetically. Added the IOCTL service `ADADIO_IOCTL_AIN_BUF_CLEAR`. Updated the usage of the Wait Event `timeout_ms` field. Updated material on the open call. Added open access mode descriptions. Updated the kernel support table.
May 4, 2016	Updated to version 4.1.66.13.0. Removed the `built` field from the `/proc` file. Updated the kernel support table.
September 16, 2015	Updated to version 4.0.60.8.0. Updated the device node name to include a period before the device index. Removed double underscore that prefaced various data types.
September 4, 2014	Updated to version 3.8.55.0.
February 28, 2014	Updated to version 3.7.52.0. Updated the kernel support data.
January 8, 2014	Updated to version 3.6.51.0. Updated the kernel support data.
November 6, 2013	Updated to version 3.6.48.0. Removed the sample application `testapp` from the release.
July 7, 2013	Updated to version 3.6.45.0. Updated the kernel support data.
July 23, 2012	Updated to version 3.6.39.0. Updated the kernel support data.
December 20, 2011	Updated to version 3.5.34.0.
November 10, 2011	Updated to version 3.4.32.0.
March 21, 2011	Updated to version 3.3.22.0. Various editorial changes. Removed the IRQ_ENABLE IOCTL service – local interrupts are always enabled. Removed the IRQ_STATUS IOCTL service. Updated the CPU and Kernel Support information. Updated the comments for the Initialize IOCTL service. Changed the spelling of various Auto Calibration related software items.
December 28, 2009	Updated to version 3.2.13.0.
July 29, 2009	Updated to version 3.1.9.0. Minor change to kernel support table.
April 17, 2009	Updated to version 3.0.5.0.
April 10, 2009	Updated to version 3.0.4.0. Extensive interface changes. Added sample applications.
September 2, 2008	Updated to version 2.1.0. Numerous modifications.
November 2, 2004	Updated to version 2.0.0.
February 11, 2003	Ported the driver to the 2.4 kernel.
February 10, 2003	The test application now uses `ADADIODocSrcLib`. Some code examples were updated.
February 7, 2003	Added notes about `mmap()` of GSC registers when they aren't on a page boundary. The documentation sources are now included as a library.
June 13, 2002	Initial release.