DIO40

Discrete 40-Bit Digital I/O

PMC64-HPDI40LS-DIO

Linux Device Driver User Manual

Manual Revision: December 7, 2016
Driver Release Version 3.0.68.18.0

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

Table of Figures

1. No table of figures entries found.Introduction

This user manual applies to driver version 3.0.68.18.0.

1.1. Purpose

The purpose of this document is to describe the interface to the DIO40 Linux device driver and the driver interface library. This software provides the interface between "Application Software" and the DIO40 board. The interface to this board is at the device level.

1.2. Acronyms

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

Acronyms	Description
DMA	Direct Memory Access
PCI	Peripheral Component Interconnect
PMC	PCI Mezzanine Card

1.3. Definitions

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

Term	Definition
Application	Application means the user mode process, which runs in the user space with user mode privileges.
DIO40	This is a substitute for the product's formal name, which is HPDI40LS-DIO. Other prefixes or suffixes may apply.
Driver	Driver means the kernel mode device driver, which runs in the kernel space with kernel mode privileges.

1.4. Software Overview

The DIO40 driver software executes under control of the Linux operating system and runs in Kernel Mode as a Kernel Mode device driver. The DIO40 device driver is implemented as a standard dynamically loadable Linux device driver written in the 'C' programming language. With the driver, user applications are able to open and close a device and, while open, perform I/O control operations.

1.5. Hardware Overview

The DIO40 is a simple 40-bit discrete I/O interface board. The host side connection is PCI based and the external I/O interface is via an 80 pin connector. The external interface includes 40 pin pairs that can each be arbitrarily programmed as either input or output. The 40 programmable pins are divided into two groups; Port A and Port B. Ports A is 8-bits wide and port B is 32-bits wide. All port pins are individually and arbitrarily programmable as inputs or outputs. The DIO40 has no DMA or interrupt functionality.

1.6. Reference Material

The following reference material may be of particular benefit in using the DIO40 and this driver. The specifications provide the information necessary for an in depth understanding of the specialized features implemented on this board.

- The applicable *HPDI40LS User Manual* from General Standards Corporation.

- The *PCI9656 PCI Bus Master Interface Chip* data handbook from PLX Technology, Inc.

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

2. Installation

2.1. CPU and Kernel Support

The driver is designed to operate with Linux kernel versions 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 was tested under the below listed kernels.

Kernel	Distribution	X86	
		32-bit	64-bit
4.5.5	Red Hat Fedora Core 24	Yes	Yes
4.2.3	Red Hat Fedora Core 23	Yes	Yes
4.0.4	Red Hat Fedora Core 22	Yes	Yes
3.17.4	Red Hat Fedora Core 21	Yes	Yes
3.11.10	Red Hat Fedora Core 20	Yes	Yes
3.9.5	Red Hat Fedora Core 19	Yes	Yes
3.6.10	Red Hat Fedora Core 18	Yes	Yes
3.3.4	Red Hat Fedora Core 17	Yes	Yes
3.1.0	Red Hat Fedora Core 16	Yes	Yes
2.6.38	Red Hat Fedora Core 15	Yes	Yes
2.6.35	Red Hat Fedora Core 14	Yes	Yes
2.6.33	Red Hat Fedora Core 13	Yes	Yes
2.6.31	Red Hat Fedora Core 12	Yes	Yes
2.6.29	Red Hat Fedora Core 11	Yes	Yes
2.6.27	Red Hat Fedora Core 10	Yes	Yes
2.6.25	Red Hat Fedora Core 9	Yes	Yes
2.6.23	Red Hat Fedora Core 8	Yes	Yes
2.6.21	Red Hat Fedora Core 7	Yes	Yes
2.6.18	Red Hat Fedora Core 6	Yes	Yes
2.6.15	Red Hat Fedora Core 5	Yes	Yes
2.6.11	Red Hat Fedora Core 4	Yes	Yes
2.6.9	Red Hat Fedora Core 3	Yes	Yes
2.4.21	Red Hat Enterprise Linux Workstation Release 3	Yes	
2.2.14	Red Hat Linux 6.2	Yes	

NOTE: While only Red Hat Fedora and Enterprise 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 on an SMP host.

2.1.1. 32-bit Support Under 64-bit Environments

This DIO40 device 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/dio40` file will be "`no`".

2.2. The /proc File System

While the driver is installed, the text file `/proc/dio40` can be read to obtain information about the driver. Each file entry includes an entry name followed immediately by a colon, a space character, then the entry value. Below is an example of what appears in the file, followed by descriptions of each entry.

```
version: 3.0.68.18
boards: 1
```

Entry	Description
`version`	This gives the driver version number in the form `x.x.x.x`.
`boards`	This identifies the total number of boards the driver detected.

2.3. File List

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

File	Description
`dio40.tar.gz`	This archive contains the driver and all related sources.
`dio40_linux_um.pdf`	This is a PDF version of this user manual.

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
`dio40`	This is the driver root directory. It contains the documentation, the overall make script and the below listed subdirectories.
`…/din`	This directory contains the Digital Input sample application.
`…/docsrc`	This directory contains the code samples from this document (section 7, page 34).
`…/dout`	This directory contains the Digital Output sample application.
`…/driver`	This directory contains the driver and its sources (section 3, page 12).
`…/led`	This directory contains the LED sample application.
`…/lib`	This directory contains the DIO40 API Library (section 3.2, page 14).
`…/sbtest`	This directory contains the Single Board Test application.
`…/utils`	This directory contains utility sources used by the sample applications.

2.5. Installation

Install the driver and its related files following the below listed steps. This includes the device driver, the interface library, the documentation source code, and the sample applications.

1. Change the current directory to `/usr/src/linux/drivers`. (The path name may vary among distributions and kernel versions.)

2. Copy the archive file `dio40.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 `dio40` in the current directory, and then copies all of the archive's files into this new directory.

   ```
   tar –xzvf dio40.tar.gz
   ```

2.6. Removal

Follow the below steps to remove the driver and its related files. This includes the device driver, the interface library, the documentation source code, and the sample applications.

1. Shutdown the driver as described in following paragraphs.

2. Change to the directory where the driver archive was installed. This should be `/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 dio40.tar.gz dio40
   ```

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

   ```
   rm –f /dev/dio40.*
   ```

5. If the automated startup procedure was adopted (described in following paragraphs), then edit the system startup script `rc.local` and remove the line that invokes the `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, and it will also load the driver. The script is named `make_all`. Follow the below steps to perform an overall make and to load the driver.

1. Change to the driver's directory, which may be `/usr/src/linux/drivers/dio40`.

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

   ```
   ./make_all
   ```

3. The Driver

This driver and its related files are contained in the archive file `dio40.tar.gz`. The archive's device driver files are listed below. The paragraphs that follow give installation, build and startup instructions.

File	Description
`*.c`	The driver source files.
`*.h`	The driver header files.
`dio40.h`	The driver interface header file. This header should be included by DIO40 applications.
`Makefile`	The driver make file.
`makefile.dep`	An automatically generated make dependency file.
`start`	Shell script to install the driver executable and device nodes.

3.1.1. Build

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

Follow the below steps to build the driver.

1. Change to the directory where the driver and its sources were installed. This should be `/usr/src/linux/drivers/dio40/driver`.

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

   ```
   make clean
   ```

3. Build the driver by issuing the below command.

   ```
   make all
   ```

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

3.1.2. Startup

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

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

3.1.2.1. Manual Driver Startup Procedures

Start the driver manually by following the below listed steps.

1. Login as root user, as some of the steps require root privileges.

2. Change to the directory where the driver was installed. This should be `/usr/src/linux/drivers/dio40/driver`.

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

   ```
   ./start
   ```

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

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

NOTE: The DIO40 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.

4. Verify that the device module has been loaded by issuing the below command and examining the output. The module name `dio40` should be included in the output.

   ```
   lsmod
   ```

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

   ```
   ls –l /dev/dio40.*
   ```

3.1.2.2. Automatic Driver Startup Procedures

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

1. Locate and edit the system startup script `rc.local`, which should be in the `/etc/rc.d` directory. Modify the file by adding the below line so that it is executed with every reboot.

   ```
   /usr/src/linux/drivers/dio40/driver/start
   ```

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

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

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

3.1.3. Verification

WARNING: When using the test application the DIO40 and any externally attached equipment may be damaged if the DIO40's external interface has a cable other than an appropriate loop back cable attached. Damage may result because the application methodically configures each I/O pin as an output and drives the output to both its high and low states. No damage will result if no cable at all is attached.

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

1. Change to the directory where the sample application `sbtest` was installed.

2. Start the application by issuing the below command.

```
./sbtest
```

3.1.4. Version

The driver version number can be obtained in a variety of ways. It is reported by the driver both when the driver is loaded and when it is unloaded (depending on kernel configuration options, this may be visible only in `/var/log/messages`). It is recorded in the text file `/proc/dio40`.

3.1.5. Shutdown

Shutdown the driver following the below listed steps.

1. Login as root user, as some of the steps require root privileges.

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

   ```
   rmmod dio40
   ```

3. Verify that the driver module has been unloaded by issuing the below command. The module name `dio40` should not be in the list.

   ```
   lsmod
   ```

3.2. Driver Interface Library

The archive file `dio40.tar.gz` contains a library with a feature based interface to the DIO40 and the driver. The purpose of the library is to simplify some of the details of using the DIO40 and the driver. The library is provided in a statically linkable form and includes all source code. These files are installed into the directory `/usr/src/linux/drivers/dio40/lib`.

File	Description
`*.c`	These are the C source files.
`dio40_lib.h`	This is the library header file.
`makefile`	This is the library make file.
`makefile.dep`	This is an automatically generated make dependency file.

3.2.1. Build

Follow the below steps to build the library.

1. Change to the directory where the library sources were installed. This should be `/usr/src/linux/drivers/dio40/lib`.

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

   ```
   make clean
   ```

3. Build the library by issuing the below command.

   ```
   make all
   ```

3.2.2. Use

The library is used both at application compile time and at application link time. Compile time use has two requirements. First, include the header file `dio40_lib.h` in each module referencing a library component. Second, expand the include file search path to search the directory where the library header is located. This should be `/usr/src/linux/drivers/dio40/lib`. Link time use also has two requirements. First, include the static library `dio40_lib.a` in the list of files to be linked into the application. Second, expand the library file search path to search the directory where the library is located. This should also be `/usr/src/linux/drivers/dio40/lib`.

4. Driver Interface

The DIO40 driver conforms to the device driver standards required by the Linux Operating System and contains the standard driver entry points. The device driver provides a standard driver interface to the GSC DIO40 board for Linux applications. The interface includes various macros, data types and functions, all of which are described in the following paragraphs. The DIO40 specific portion of the driver interface is defined in the header file `dio40.h`, portions of which are described in this section. The header defines numerous items in addition to those described here.

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

4.1. Macros

The driver interface includes the following macros which are defined in `dio40.h`. The header also contains various other utility type macros which are provided without documentation.

4.1.1. IOCTL

The IOCTL macros are documented following the function call descriptions.

4.1.2. Registers

4.1.2.1. GSC Registers

The following table gives the complete set of GSC specific DIO40 registers. For detailed definitions of these registers refer to the *DIO40 User Manual*.

Macros	Description
`DIO40_GSC_BCR`	Board Control Register (BCR)
`DIO40_GSC_BSR`	Board Status Register (BSR)
`DIO40_GSC_CCAR`	Cable Control A Register (CCAR)
`DIO40_GSC_CCBR`	Cable Control B Register (CCBR)
`DIO40_GSC_CIAR`	Cable Input A Register (CIAR)
`DIO40_GSC_CIBR`	Cable Input B Register (CIBR)
`DIO40_GSC_COAR`	Cable Output A Register (COAR)
`DIO40_GSC_COBR`	Cable Output B Register (COBR)
`DIO40_GSC_FRR`	Firmware Revision Register (FRR)
`DIO40_GSC_LOR`	LED Output Register (LOR)

4.1.2.2. PLX PCI 9656 Registers

The PCI interface chip used by the DIO40 is a PLX PCI9656. As this chip's registers are of little use to DIO40 users, the PCI and PLX feature registers are not listed here. For detailed definitions of these registers refer to the *PCI9656 Data Book*.

4.2. Data Types

This driver interface includes the following data types which are defined in `dio40.h`.

4.2.1. gsc_reg_t

This structure defines the data fields for the information involved in the register access IOCTL services. Read the details of the individual services for additional information.

Definition

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

Fields	Description
reg	This field identifies the register to be accessed.
value	This field identifies the value retrieved by read operations and the value to apply by write operations.
mask	This field identifies the register bits from the value field that are to be applied during the read-modify-write IOCTL service. If a bit is set in the mask, then the corresponding value bit is applied to the register. If a mask bit is not set then the corresponding register bit is left unchanged.

4.3. Functions

This driver interface includes the following functions.

4.3.1. close()

This function is the entry point to close a connection to an open DIO40 board. This function should only be called after a successful open of the respective device.

> **NOTE:** The functionality of the close() system call is available in the DIO40 Library via the dio40_close() function. See section 5.6.1 on page 31.

Prototype

```
int close(int fd);
```

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

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

Example

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

#include "dio40_dsl.h"

int dio40_dsl_close(int fd)
{
    int status;

    status  = close(fd);
```

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

    return(status);
}
```

4.3.2. ioctl()

This function is the entry point to performing setup and control operations on a DIO40 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 any additional arguments. The set of supported IOCTL services is defined in a following section.

> **NOTE:** The functionality of the `ioctl()` system call is available in the DIO40 Library via the `dio40_ioctl()` function. See section 5.6.2 on page 31.

Prototype

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

Argument	Description
`fd`	This is the file descriptor of the device to access.
`request`	This specifies the desired operation to be performed.
`...`	This is any additional arguments. If `request` does not call for any additional arguments, then any additional arguments provided are ignored. The DIO40 IOCTL services use at most one argument, which is represented by a 32-bit value.

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

Example

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

#include "dio40_dsl.h"

int dio40_dsl_ioctl(int fd, int request, void *arg)
{
    int status;

    status  = ioctl(fd, request, (unsigned long) arg);

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

    return(status);
}
```

4.3.3. open()

This function is the entry point to open a connection to an DIO40 board. The pathname to an DIO40 board is `/dev/dio40.`*n*, where the trailing "*n*" is the zero based index of the board to access.

NOTE: The functionality of the `open()` system call is available in the DIO40 Library via the `dio40_open()` function. See section 5.6.4 on page 32. This function permits an application to access a DIO40 by specifying only the index of the board to access. The remaining details are handled by the library.

Prototype

```
int open(const char* pathname, int flags);
```

Argument	Description
`pathname`	This is the name of the device to open.
`flags`	This is the desired device access. The option `O_RDWR` is required. The option `O_APPEND` is optional and opens the device for shared access. This permits multiple applications to gain simultaneous access to the same device. If this flag is omitted, then the request is for exclusive access by the calling process.

NOTE: Upon successful opening, either for exclusive access or the initial shared access, the device and all settings are put in an initialized state. The device state is unaltered when a shared access request gains access to a device that is already open.

NOTE: Another form of the `open()` function has a `mode` argument. This form is not displayed here as the `mode` argument is ignored when opening an existing file/device.

Return Value	Description
`-1`	An error occurred. Consult `errno`.
else	A valid file descriptor.

Example

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

#include "dio40_dsl.h"

int dio40_dsl_open(int index, int share)
{
    int     fd;
    int     flags;
    char    name[80];

    sprintf(name, "/dev/" DIO40_BASE_NAME ".%d", index);
    flags   = O_RDWR | (share ? O_APPEND : 0);
    fd      = open(name, flags);

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

    return(fd);
}
```

4.3.3.1. Access Modes

The presence or absence of the `O_APPEND` flag in the open call determines the device access mode, as follows.

Shared Access Mode:

In the `open()` call, including the `O_APPEND` flag opens the device in Shared Access Mode. The first `open()` call including this flag will succeed and return with the device in an initialized state. Subsequent `open()` calls, if this flag is present, will also succeed, but will not alter the device state. Once opened in Shared Access Mode, device access remains in this mode until all Shared Access Mode open requests release the device with a `close()` call.

Exclusive Access Mode:

In the `open()` call, excluding the `O_APPEND` flag opens the device in Exclusive Access Mode. In this mode, only one application at a time can access the device. The first `open()` call will succeed and will return with the device in an initialized state. Subsequent `open()` calls, whether or not the `O_APPEND` flag is used, will fail until the device is released with a `close()` call by the initial process.

4.3.4. read()

The DIO40 does not support a read operation as the board has neither data storage nor synchronous data reception capability. Data read operations are performed by calling the appropriate library interface routines or reading the appropriate register.

4.3.5. write()

The DIO40 does not support a write operation as the board has neither data storage nor synchronous data transmission capability. Data write operations are performed by calling the appropriate library interface routines or writing to the appropriate register.

4.4. IOCTL Services

The DIO40 driver implements the following IOCTL services. Each service is described along with the applicable `ioctl()` function arguments. In the definitions given the optional argument is identified as `arg` and is an `unsigned long` data type. Unless otherwise stated the return value definitions are those defined for the `ioctl()` function call and any errors codes are accessed via `errno`.

4.4.1. DIO40_IOCTL_REG_MOD

This service performs a read-modify-write operation on a DIO40 register. This includes only the GSC specific registers. All PCI and PLX PCI9656 feature set registers are read-only. Refer to `dio40.h` for a complete list of the accessible registers.

Usage

ioctl() Argument	**Description**
request	DIO40_IOCTL_REG_MOD
arg	gsc_reg_t*

Example

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

```
#include "dio40_dsl.h"

int dio40_dsl_reg_mod(
    int fd,
    u32 reg,
    u32 value,
    u32 mask)
{
    gsc_reg_t  parm;
    int        status;

    parm.reg   = reg;
    parm.value = value;
    parm.mask  = mask;
    status     = ioctl(fd,
                       DIO40_IOCTL_REG_MOD,
                       (unsigned long) &parm);

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

    return(status);
}
```

4.4.2. DIO40_IOCTL_REG_READ

This service reads the value of a DIO40 register. This includes all PCI registers, all PLX PCI9656 feature set registers, and all GSC specific registers. Refer to `dio40.h` for a complete list of the accessible registers.

Usage

ioctl() Argument	**Description**
request	DIO40_IOCTL_REG_READ
arg	gsc_reg_t*

Example

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

#include "dio40_dsl.h"

int dio40_dsl_reg_read(int fd, u32 reg, u32* value)
{
    gsc_reg_t  parm;
    int        status;

    parm.reg   = reg;
    parm.value = (u32) 0xDEADBEEF;
    parm.mask  = 0;    // ignored for reads
    status     = ioctl(fd,
                       DIO40_IOCTL_REG_READ,
                       (unsigned long) &parm);
```

```
    if (value)
        value[0]    = parm.value;

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

    return(status);
}
```

4.4.3. DIO40_IOCTL_REG_WRITE

This service writes a value to a DIO40 register. This includes only the GSC specific registers. All PCI and PLX PCI9656 feature set registers are read-only. Refer to `dio40.h` for a complete list of the accessible registers.

Usage

ioctl() Argument	**Description**
request	DIO40_IOCTL_REG_WRITE
arg	gsc_reg_t*

Example

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

#include "dio40_dsl.h"

int dio40_dsl_reg_write(int fd, u32 reg, u32 value)
{
    gsc_reg_t   parm;
    int         status;

    parm.reg    = reg;
    parm.value  = value;
    parm.mask   = 0;    // ignored for writes
    status      = ioctl(fd,
                        DIO40_IOCTL_REG_WRITE,
                        (unsigned long) &parm);

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

    return(status);
}
```

5. Driver Interface Library

The Driver Interface Library is provided to simplify access to some of the DIO40 features. The library is provided as a statically linkable library and includes full source code. The purpose of this section is merely to give a summary of the library's interface functions. The library interface is defined in the header file `dio40_lib.h`. Application sources need to include this header to use the listed services.

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

5.1. GPIO Port A Services

The following services provide access to GPIO Port A.

5.1.1. dio40_gpio_a_dir_get()

This function retrieves the direction for the Port A pins. If a bit is set, then the port pin is an output. Otherwise it is an input. Bit zero corresponds to Port A0.

Prototype

```
int dio40_gpio_a_dir_get(int fd, u8* dir);
```

Argument	Description
`fd`	This is the file descriptor of the device to be accessed.
`dir`	If non-NULL, the direction data is stored here.

Return Value	Description
`0`	The operation succeeded.
else	The value from `errno` for the error that occurred.

5.1.2. dio40_gpio_a_dir_mod()

This function performs a read-modify-write operation on the direction settings for the Port A pins. If a bit is set, then the port pin is an output. Otherwise it is an input. Bit zero corresponds to Port A0.

Prototype

```
int dio40_gpio_a_dir_mod(int fd, u8 dir, u8 mask);
```

Argument	Description
`fd`	This is the file descriptor of the device to be accessed.
`dir`	This is the direction data to apply.
`mask`	These are the direction bits to modify. If a bit is set here, then the direction bit is modified. If the bit is clear here, the direction bit is unaltered.

Return Value	Description
`0`	The operation succeeded.
else	The value from `errno` for the error that occurred.

5.1.3. dio40_gpio_a_dir_set()

This function updated the direction settings for the Port A pins. If a bit is set, then the port pin is an output. Otherwise it is an input. Bit zero corresponds to Port A0.

Prototype

```
int dio40_gpio_a_dir_set(int fd, u8 dir);
```

Argument	Description
fd	This is the file descriptor of the device to be accessed.
dir	This is the direction data to apply.

Return Value	Description
0	The operation succeeded.
else	The value from errno for the error that occurred.

5.1.4. dio40_gpio_a_in_get()

This function reads the input from the cable's Port A pins. Bit zero corresponds to Port A0.

Prototype

```
int dio40_gpio_a_in_get(int fd, u8* data);
```

Argument	Description
fd	This is the file descriptor of the device to be accessed.
data	The data read is recorded here, if non-NULL.

Return Value	Description
0	The operation succeeded.
else	The value from errno for the error that occurred.

5.1.5. dio40_gpio_a_out_get()

This function retrieves what is recorded for output on the Port A pins. Bit zero corresponds to Port A0.

Prototype

```
int dio40_gpio_a_out_get(int fd, u8* data);
```

Argument	Description
fd	This is the file descriptor of the device to be accessed.
data	If non-NULL, the data is stored here.

Return Value	Description
0	The operation succeeded.
else	The value from errno for the error that occurred.

5.1.6. dio40_gpio_a_out_mod()

This function performs a read-modify-write on what is recorded for output on the Port A pins. Bit zero corresponds to Port A0.

Prototype

```
int dio40_gpio_a_out_mod(int fd, u8 data, u8 mask);
```

Argument	Description
fd	This is the file descriptor of the device to be accessed.
data	This is the data to apply.
mask	These are the data bits to modify. If a bit is set here, then the recorded bit is modified. If the bit is clear here, the recorded bit is unaltered.

Return Value	Description
0	The operation succeeded.
else	The value from errno for the error that occurred.

5.1.7. dio40_gpio_a_out_set()

This function updates what is recorded for output on the Port A pins. Bit zero corresponds to Port A0.

Prototype

```
int dio40_gpio_a_out_set(int fd, u8 data);
```

Argument	Description
fd	This is the file descriptor of the device to be accessed.
data	This is the data to apply.

Return Value	Description
0	The operation succeeded.
else	The value from errno for the error that occurred.

5.2. GPIO Port A Tx Clock Services

The following services provide access to GPIO Port A Tx Clock features.

5.2.1. dio40_gpio_a0_tx_clock_get()

This function retrieves the enabled state of the Port A0 Tx Clock feature. If the feature is enabled, then the returned value is one. If the feature is disabled, then the returned value is zero.

Prototype

```
int dio40_gpio_a0_tx_clock_get(int fd, u8* get);
```

Argument	Description
fd	This is the file descriptor of the device to be accessed.
get	If non-NULL, the enabled state is stored here.

Return Value	Description
0	The operation succeeded.
else	The value from errno for the error that occurred.

5.2.2. dio40_gpio_a0_tx_clock_set()

This function sets the enabled state of the Port A0 Tx Clock feature.

Prototype

```
int dio40_gpio_a0_tx_clock_set(int fd, int set);
```

Argument	Description
fd	This is the file descriptor of the device to be accessed.
set	The feature is enabled if this is non-zero, and it is disabled if this is zero.

Return Value	Description
0	The operation succeeded.
else	The value from errno for the error that occurred.

5.2.3. dio40_gpio_a7_tx_clock_get()

This function retrieves the enabled state of the Port A7 Tx Clock feature. If the feature is enabled, then the returned value is one. If the feature is disabled, then the returned value is zero.

Prototype

```
int dio40_gpio_a7_tx_clock_get(int fd, u8* get);
```

Argument	Description
fd	This is the file descriptor of the device to be accessed.
get	If non-NULL, the enabled state is stored here.

Return Value	Description
0	The operation succeeded.
else	The value from errno for the error that occurred.

5.2.4. dio40_gpio_a7_tx_clock_set()

This function sets the enabled state of the Port A0 Tx Clock feature.

Prototype

```
int dio40_gpio_a7_tx_clock_set(int fd, int set);
```

Argument	Description
fd	This is the file descriptor of the device to be accessed.
set	The feature is enabled if this is non-zero, and it is disabled if this is zero.

Return Value	Description
0	The operation succeeded.
else	The value from errno for the error that occurred.

5.3. GPIO Port B Services

The following services provide access to GPIO Port B.

5.3.1. dio40_gpio_b_dir_get()

This function retrieves the direction for the Port B pins. If a bit is set, then the port pin is an output. Otherwise it is an input. Bit zero corresponds to Port B0.

Prototype

```
int dio40_gpio_b_dir_get(int fd, u32* dir);
```

Argument	Description
fd	This is the file descriptor of the device to be accessed.
dir	If non-NULL, the direction data is stored here.

Return Value	Description
0	The operation succeeded.
else	The value from errno for the error that occurred.

5.3.2. dio40_gpio_b_dir_mod()

This function performs a read-modify-write operation on the direction settings for the Port B pins. If a bit is set, then the port pin is an output. Otherwise it is an input. Bit zero corresponds to Port B0.

Prototype

```
int dio40_gpio_b_dir_mod(int fd, u32 dir, u32 mask);
```

Argument	Description
fd	This is the file descriptor of the device to be accessed.
dir	This is the direction data to apply.
mask	These are the direction bits to modify. If a bit is set here, then the direction bit is modified. If the bit is clear here, the direction bit is unaltered.

Return Value	Description
0	The operation succeeded.
else	The value from errno for the error that occurred.

5.3.3. dio40_gpio_b_dir_set()

This function updated the direction settings for the Port B pins. If a bit is set, then the port pin is an output. Otherwise it is an input. Bit zero corresponds to Port B0.

Prototype

```
int dio40_gpio_b_dir_set(int fd, u32 dir);
```

Argument	Description
fd	This is the file descriptor of the device to be accessed.
dir	This is the direction data to apply.

Return Value	Description
0	The operation succeeded.
else	The value from errno for the error that occurred.

5.3.4. dio40_gpio_b_in_get()

This function reads the input from the cable's Port B pins. Bit zero corresponds to Port B0.

Prototype

```
int dio40_gpio_b_in_get(int fd, u32* data);
```

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

data	The data read is recorded here, if non-NULL.

Return Value	Description
0	The operation succeeded.
else	The value from errno for the error that occurred.

5.3.5. dio40_gpio_b_out_get()

This function retrieves what is recorded for output on the Port B pins. Bit zero corresponds to Port B0.

Prototype

```
int dio40_gpio_b_out_get(int fd, u32* data);
```

Argument	Description
fd	This is the file descriptor of the device to be accessed.
data	If non-NULL, the data is stored here.

Return Value	Description
0	The operation succeeded.
else	The value from errno for the error that occurred.

5.3.6. dio40_gpio_b_out_mod()

This function performs a read-modify-write on what is recorded for output on the Port B pins. Bit zero corresponds to Port B0.

Prototype

```
int dio40_gpio_b_out_mod(int fd, u32 data, u32 mask);
```

Argument	Description
fd	This is the file descriptor of the device to be accessed.
data	This is the data to apply.
mask	These are the data bits to modify. If a bit is set here, then the recorded bit is modified. If the bit is clear here, the recorded bit is unaltered.

Return Value	Description
0	The operation succeeded.
else	The value from errno for the error that occurred.

5.3.7. dio40_gpio_b_out_set()

This function updates what is recorded for output on the Port B pins. Bit zero corresponds to Port B0.

Prototype

```
int dio40_gpio_b_out_set(int fd, u32 data);
```

Argument	Description
fd	This is the file descriptor of the device to be accessed.
data	This is the data to apply.

Return Value	Description
0	The operation succeeded.
else	The value from errno for the error that occurred.

5.4. LED Services

The following services provide access to LEDs on the back of the DIO40.

5.4.1. dio40_led_get()

This function retrieves the state of the LEDs. If a bit is set, then the LED is on. Otherwise it is off.

Prototype

```
int dio40_led_get(int fd, u8* on);
```

Argument	Description
fd	This is the file descriptor of the device to be accessed.
on	If non-NULL, the state data is stored here.

Return Value	Description
0	The operation succeeded.
else	The value from errno for the error that occurred.

5.4.2. dio40_led_mod()

This function performs a read-modify-write on the LEDs' on/off state. If a bit is set, then the LED is on. Otherwise it is off.

Prototype

```
int dio40_led_mod(int fd, u8 on, u8 mask);
```

Argument	Description
fd	This is the file descriptor of the device to be accessed.
on	This is the data to apply.
mask	These are the data bits to modify. If a bit is set here, then the LED bit is modified. If the bit is clear here, the LED bit is unaltered.

Return Value	Description
0	The operation succeeded.
else	The value from errno for the error that occurred.

5.4.3. dio40_led_set()

This function updates what is on/off state of the LEDs. If a bit is set, then the LED is on. Otherwise it is off.

Prototype

```
int dio40_led_set(int fd, u8 on);
```

Argument	Description
fd	This is the file descriptor of the device to be accessed.
on	This is the data to apply.

Return Value	Description
0	The operation succeeded.
else	The value from errno for the error that occurred.

5.5. Register Access Services

The following services provide access to the DIO40 registers.

5.5.1. dio40_reg_mod()

This function performs a read-modify-write on a specified DIO40 register. This applies to firmware registers only, as all PCI and PLX registers are read-only.

Prototype

```
int dio40_reg_mod(int fd, u32 reg, u32 value, u32 mask);
```

Argument	Description
fd	This is the file descriptor of the device to be accessed.
reg	This identifies the register to be read.
Value	This is the value to apply.
mask	These are the value bits to modify. If a bit is set here, then the register bit is modified. If the bit is clear here, the register bit is unaltered.

Return Value	Description
0	The operation succeeded.
else	The value from errno for the error that occurred.

5.5.2. dio40_reg_read()

This function reads a specified DIO40 registers.

Prototype

```
int dio40_reg_read(int fd, u32 reg, u32* value);
```

Argument	Description
fd	This is the file descriptor of the device to be accessed.
reg	This identifies the register to be read.
value	If non-NULL, the value read is recorded here.

Return Value	Description
0	The operation succeeded.
else	The value from errno for the error that occurred.

5.5.3. dio40_reg_write()

This function updates the value of a specified register. This applies to firmware registers only, as all PCI and PLX registers are read-only.

Prototype

```
int dio40_reg_write(int fd, u32 reg, u32 value);
```

Argument	Description
fd	This is the file descriptor of the device to be accessed.
reg	This identifies the register to be accessed.
value	This is the value to apply.

Return Value	Description
0	The operation succeeded.
else	The value from errno for the error that occurred.

5.6. Additional Services

The following are additional miscellaneous services.

5.6.1. dio40_close()

This function closes an open connection to a DIO40.

Prototype

```
int dio40_close(int fd);
```

Argument	Description
fd	This is the file descriptor of the device to close.

Return Value	Description
0	The operation succeeded.
else	The value from errno for the error that occurred.

5.6.2. dio40_ioctl()

This function issues an IOCTL call for a connection to a DIO40.

Prototype

```
int dio40_ioctl(int fd, int request, void *arg);
```

Argument	Description
fd	This is the file descriptor of the device to access.
request	This is the code for the service to perform. See section 4.4 starting on page 20.
arg	This is the service specific argument.

Return Value	Description
0	The operation succeeded.
else	The value from errno for the error that occurred.

5.6.3. dio40_lib_version()

This function returns the library version number and build date and time.

Prototype

```
void dio40_lib_version(
    char*   ver,
    size_t  v_size,
```

```
    char*   built,
    size_t  b_size);
```

Argument	Description
ver	The version number is returned her in the form of "X.X.X".
v_size	This is the size of the above buffer.
built	The library build date and time is returned here in the C form of sprintf("%s, %s", __DATE__, __TIME__).
b_size	This is the size of the above buffer.

5.6.4. dio40_open()

This function opens a connection to a specified DIO40. The desired board is specified by the zero based index, where the first board in the system is index zero.

Prototype

```
int dio40_open(int board);
```

Argument	Description
board	This is the index of the board to access.

Return Value	Description
else	The file descriptor to use as the operation succeeded.
-1	An error occurred. Refer to errno for the specific error condition.

5.6.5. dio40_reset()

This function reset the entire DIO40.

Prototype

```
int dio40_reset(int fd);
```

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

Return Value	Description
0	The operation succeeded.
else	The value from errno for the error that occurred.

6. Operating Information

This section explains some basic operational procedures for using the DIO40. This is in no way intended to be a comprehensive guide. This is simply to address a very few issues relating to their use.

No additional information is available at this time.

7. 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.

7.1. 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.
`docsrc/dio40_dsl.a`	This is the statically linkable library file.
`docsrc/dio40_dsl.h`	This is the primary utility header file.

7.2. Build

The library is built via the Overall Make Script (section 2.7, page 11), but can be built separately following the below steps.

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

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 file with the objects being linked with the application.

File	Default Location
`dio40_dsl.h`	`/usr/src/linux/drivers/dio40/docsrc`
`dio40_dsl.a`	`/usr/src/linux/drivers/dio40/docsrc`

8. 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.

8.1. Files

The library files are summarized in the table below.

File	Description
`utils/util_*.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.
`utils/dio40_utils.a`	This is the statically linkable library file.
`utils/dio40_utils.h`	This is the primary utility header file.

8.2. Build

The library is built via the Overall Make Script (section 2.7, page 11), but can be built separately following the below steps.

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

2. Remove all 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 all
   ```

8.3. 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.

File	Default Location
`dio40_utils.h`	`/usr/src/linux/drivers/dio40/utils`
`dio40_utils.a`	`/usr/src/linux/drivers/dio40/utils`

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, 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. din - Digital Input

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

9.2. dout - Digital Output - …/dout/

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

9.3. led – LED Exerciser - …/led/

This application exercises the board LEDs.

9.4. 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
December 7, 2016	Updated to release version 3.0.68.18.0. Updated the device node name to include a period before the device index. Removed double underscore that prefaced various data types. Removed the `built` field from the `/proc` file. Updated the kernel support table. Updated material on the open call. Added open access mode descriptions. Added a section for general operating information. Made various miscellaneous updates. Some document reorganization.
October 16, 2014	Updated to version 2.0.57.0. Updated the kernel support table. Modified driver interface. Renamed data structure `dio40_reg_t` to `gsc_reg_t`. Deleted data structure `dio40_driver_info_t` and IOCTL service `DIO40_IOCTL_DRIVER_INFO_GET`.
November 18, 2013	Updated to version 1.2.50.0. Various editorial changes. Updated the CPU and Kernel Support information. Changed the name of the application `rx` to `din`. Changed the name of the application `tx` to `dout`. Pre-built targets are no longer provided. Updated the CPU support data.
August 29, 2006	Updated to version 1.01.0. Updated to support the 64-bit kernels and more recent 2.6 kernels.
April 7, 2006	Initial release.