OPTO32A

24 Input Bit, 8 Output Bit Optical Isolator Board

PMC-OPTO32A

Linux Device Driver
User Manual

Manual Revision: July 15, 2005

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

Preface

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

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

Table of Contents

1. Introduction

1.1. Purpose

The purpose of this document is to describe the interface to the OPTO32A Linux device driver. The driver software provides the interface between Application Software and the OPTO32A board. The driver is supplied with a sample application program.

1.2. Acronyms

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

Acronyms	Description
DMA	Direct Memory Access
GSC	General Standards Corporation
PCI	Peripheral Component Interconnect
PMC	PCI Mezzanine Card

1.3. Definitions

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

Term	Definition
Driver	Driver means the kernel mode device driver, which runs in the kernel space with kernel mode privileges.
Application	Application means the user mode process, which runs in the user space with user mode privileges.

1.4. Software Overview

The OPTO32A driver software executes under control of the Linux operating system and runs in Kernel Mode as a Kernel Mode device driver. The OPTO32A device driver is implemented as a standard dynamically loadable Linux device driver written in the C programming language. The driver allows user applications to: open, close, read, write and perform I/O control operations.

1.5. Hardware Overview

See the hardware manual for the board version for details on the hardware. Current board manual PDF files may be found at:

http://www.generalstandards.com/

Look under the “device user manuals” heading and select your board model.

1.6. Reference Material

The following reference material may be of particular benefit in using the OPTO32A and this driver. The specifications provide the information necessary for an in depth understanding of the specialized features implemented on this board.

- The applicable *OPTO32A User Manual* from General Standards Corporation.

- The PCI9080 PCI Bus Master Interface Chip data handbook from PLX Technology, Inc.

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

2. Installation

2.1. CPU and Kernel Support

The driver is designed to operate with Linux kernel version 2.4 running on a PC system with Intel x86 processor(s). Testing was performed under Red Hat Linux with kernel versions 2.2.18-14 and 2.6.x on a PC system with dual Intel x86 processors. Support for version 2.2 of the kernel has been left in the driver, but has not been tested.

NOTES:

- The driver may have to be rebuilt before being used due to kernel version differences between the GSC build host and the customer's target host.
- The driver has not been tested with a non-versioned kernel.
- The driver has only been tested on an SMP host. SMP testing is much more rigorous than single CPU systems, and helps to ensure reliability on single CPU systems.

2.2. The /proc File System

While the driver is installed, the text file `/proc/gsc_opto32a` can be read to obtain information about the driver. Each file entry includes an entry name followed immediately by a colon, a space character, and the entry value. Below is an example of what appears in the file, followed by descriptions of each entry. Note that with a debug build, there may be more information in the file.

```
version: 1.1
built: July 13 2004, 09:08:07

boards: 1
```

Entry	Description
`Version`	The driver version number in the form `x.xx`.
`Built`	The drivers build date and time as a string. It is given in the C form of `printf("%s, %s", __DATE__, __TIME__)`.
`Boards`	The total number of boards the driver detected.

2.3. File List

See the README.TXT file in the release tar for the latest file list.

This section discusses unpacking, building, installing and running the driver.

2.3.1. Installation

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

1. Create and change to the directory where you would like to install the driver source, such as `/usr/src/linux/drivers`.
2. Copy the `gsc_opto32a.tar.gz` file into the current directory. The actual name of the file may be different depending on the release version.

3. Issue the following command to decompress and extract the files from the provided archive. This creates the directory `gsc_opto32a_release` in the current directory, and then copies all of the archive's files into this new directory.

   ```
   tar –xzvf gsc_OPTO32A.tar.gz
   ```

2.3.2. Build

To build the driver:

1. Change to the directory where the driver and its sources were installed in the previous step. Remove all existing build targets by issuing the command:

   ```
   make clean
   ```

2. Edit Makefile to ensure that the KERNEL_DIR environment variable points to the correct root of the source tree for your version on Linux. The driver build uses different header versions than an application build, which is why this step is necessary. The default should be correct for 2.4 and newer kernels.

3. Build the driver by issuing the command:

   ```
   make all
   ```

NOTE: Due to the differences between the many Linux distributions some build errors may occur. The most likely cause is not having the kernel sources installed properly. See the documentation for your release of Linux for instructions on how to install the kernel sources.

To build the test applications:

1. Type the command:

   ```
   make –f app.mak
   ```

2.3.3. Startup

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

2.3.3.1. Manual Driver Startup Procedures

Start the driver manually by following the below listed steps.

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

2. Change to the directory where the driver was installed. In this example, this would be `/usr/src/linux/drivers/gsc_opto32a_release.`

3. Type:

```
./gsc_start
```

The script assumes that the driver be installed in the same directory as the script, and that the driver filename has not been changed from that specified in Makefile. The above step must be repeated each time the host is rebooted. It is possible to have the script run at system startup. See below for instructions on automatically starting the driver.

NOTE: The kernel assigns the OPTO32A device node major number dynamically. The minor numbers and the device node suffix numbers are index numbers beginning with zero, and increase by one for each additional board installed.

4. Verify that the device module has been loaded by issuing the below command and examining the output. The module name `gsc_opto32a` 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/gsc_opto32a*
   ```

2.3.3.2. Automatic Driver Startup Procedures

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

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

   ```
   /usr/src/linux/drivers/gsc_opto32a_release/gsc_start
   ```

NOTE: The script assumes the driver is in the same directory as the script. Change the path as required to point to the actual location of the driver.

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

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

2.3.4. Verification

To verify that the hardware and driver are installed properly and working, the steps are:

1. Install the sample applications, if they were not installed as part of the driver install.

2. Change to the directory where the sample application `testapp` was installed.

3. Start the sample application by issuing the below command. The argument identifies which board to access. The argument is the zero based index of the board to access.

   ```
   ./testapp <board>
   ```

   So for a single-board installation, type:

```
./ testapp 0
```

The test application is described in greater detail in a later section.

2.3.5. Version

The driver version number can be obtained in a variety of ways. It is appended to the system log when the driver is loaded or unloaded (type `dmesg` to view the contents of the system log file). It is recorded in the text file `/proc/gsc_opto32a`. It is also in the driver source header file `internals.h`, which is where the version number is maintained.

2.3.6. Shutdown

Shutdown the driver following the below listed steps.

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

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

   ```
   rmmod gsc_opto32a
   ```

3. Verify that the driver module has been unloaded by issuing the below command. The module name `gsc_opto32a` should not be in the list.

   ```
   lsmod
   ```

2.3.7. Removal

Follow the below steps to remove the driver.

1. Shutdown the driver as described in the previous paragraphs.

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

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

   ```
   rm –rf gsc_opto32a.tar.gz gsc_opto32a_release
   ```

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

   ```
   rm –f /dev/gsc_opto32a*
   ```

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

2.4. Sample Application

The archive file `gsc_opto32a.tar.gz` contains a sample application. The test application is a Linux user mode application whose purpose is to demonstrate the functionality of the driver with an installed board. They are delivered undocumented and unsupported. They can however be used as a starting point for developing applications on top of the Linux driver and to help ease the learning curve. The principle application is described in the following paragraphs.

2.4.1. testapp

This sample application provides a command line driven Linux application that tests the functionality of the driver and a user specified OPTO32A board. It can be used as the starting point for application development on top of the OPTO32A Linux device driver. The application performs an automated test of the driver features. The application includes the below listed files.

File	**Description**
`testapp.c`	The test application source file.
`testapp`	The pre-built sample application.
`app.mak`	The build script for the sample application.

2.4.2. Installation

The test application is normally installed as part of the driver install, in the same directory as the driver.

2.4.3. Build

The test applications require different header files than the driver, consequently they require a separate make script. Follow the below steps to build/rebuild the sample application.

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

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

   ```
   make –f app.mak clean
   ```

3. Build the sample applications by issuing the below command.

   ```
   Make –f app.mak -lpthread
   ```

NOTE: The build procedure assumes the driver header files are located in the current directory.

2.4.4. Execute

Follow the below steps to execute the sample application.

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

2. Start the sample application by issuing the command given below. The argument specifies the index of the board to access. Use 0 (zero) if only one board is installed.

   ```
   ./testapp 0
   ```

The test application spawns three threads that wait for input events. Fully testing the application and driver requires a loopback board that ties the input channels to the output channels, along with a pullup resistor for each channel. However, the test application does serve a good starting point.

2.4.5. Removal

```
        The sample application is removed when the driver is removed.
```

2.4.6. Driver Interface

The OPTO32A driver conforms to the device driver standards required by the Linux Operating System and contains the standard driver entry points. The device driver provides a uniform driver interface to the OPTO32A family of boards for Linux applications. The interface includes various macros, data types and functions, all of which are described in the following paragraphs. The OPTO32A specific portion of the driver interface is defined in the header file `opto32a_ioctl.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.

2.5. Macros

The driver interface includes the following macros, which are defined in `opto32a_ioctl.h`. The header also contains various other utility type macros, which are provided without documentation.

2.5.1. IOCTL

The IOCTL macros are the primary means to change the settings and configuration of the hardware. The IOCTLs are documented following the function call descriptions.

2.5.2. Registers

The following tables give the complete set of OPTO32A registers. The tables are divided by register categories. Unless otherwise stated, all registers are accessed as 32-bits. The only exception is the PCICCR register, which is 24-bits wide but accessed as if it were 32-bits wide. In this instance the upper eight-bits are to be ignored. Register values are passed as 32-bit entities and bits outside the register's native size are ignored.

2.5.2.1. GSC Registers

The following table gives the complete set of GSC specific OPTO32A registers. For detailed definitions of these registers refer to the relevant OPTO32A User Manual. The macro defines of the registers are located in opto32a_ioctl.h. Note that the hardware manual defines the register address in 8-bit address space. The driver maps the registers in 32-bit space. For example, the `OUTPUT_DATA_REG` register has local address 0x1C as defined in the hardware manual. The driver accesses this register at local address 7 (0x1C/4).

BOARD_STATUS_REG
BOARD_CONTROL_REG
RECEIVED_DATA_REG
STATE_CHANGE_REG
RECEIVE_EVENT_COUNT_REG
COS_INTERRUPT_ENABLE_REG
COS_POLARITY_REG
CLOCK_DIVISION_REG
OUTPUT_DATA_REG

2.5.2.2. PCI Configuration Registers

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

2.6. Data Types

This driver interface includes the following data types, which are defined in `opto32a_ioctl.h`.

2.6.1. device_register_params

This structure is used to transfer register data. The IOCTL_DEVICE_READ_REGISTER, and IOCTL_DEVICE_WRITE_REGISTER ioctls use this structure to read and write a user selected register. 'eRegister' stores the index of the register, range 0-`LAST_REG`, and 'ulValue' stores the register value being written or read. The absolute range for 'ulValue' is 0x0-0xFFFFFFFF, and the actual range depends on the register accessed.

Definition

```
typedef struct device_register_params {
    unsigned int eRegister;
    unsigned long ulValue;
} DEVICE_REGISTER_PARAMS, *PDEVICE_REGISTER_PARAMS;
```

Fields	Description
`eRegister`	Register to read or write. See `opto32a_ioctl.h` for register definitions.
`ulValue`	Value read from, or written to above register.

2.6.2. device_timeout_params

This structure is used to set the timeouts for the various wait functions.

Definition

```
typedef struct device_timeout_params {
    __u32 event;
    __u32 timeout;
} DEVICE_TIMEOUT_PARAMS, *PDEVICE_TIMEOUT_PARAMS;
```

Fields	Description
`event`	The index of the event for which the timeout is being set.
`timeout`	The timeout, in seconds.

2.7. Functions

This driver interface includes the following functions.

2.7.1. open()

This function is the entry point to open a handle to an OPTO32A board.

Prototype

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

Argument	Description
`pathname`	This is the name of the device to open.
`flags`	This is the desired read/write access. Use `O_RDWR`.

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

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

Example

```
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include “opto32a_ioctl.h”

int OPTO32A_open(unsigned int board)
{
    int     fd;
    char    name[80];

    sprintf(name, "/dev/gsc_opto32a%u", board);
    fd  = open(name, O_RDWR);

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

    return(fd);
}
```

2.7.2. read()

The `read()` function is not supported. A call to the `read()` function will return an error.

2.7.3. write()

The `write()` function is not supported. A call to the `write()` function will return an error,

2.7.4. close()

Close the handle to the device.

Prototype

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

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

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

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include “opto32a_ioctl.h”

int OPTO32A_close(int fd)
{
    int status;

    status  = close(fd);

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

    return(status);
}
```

2.8. IOCTL Services

This function is the entry point to performing setup and control operations on a OPTO32A board. This function should only be called after a successful open of the device. The general form of the `ioctl` call is:

```
int ioctl(int fd, int command);
```

or

```
int ioct(int fd, int command, arg*);
```

where:

`fd`	File handle for the driver. Returned from the open() function.
`command`	The command to be performed.
`arg*`	(optional) pointer to parameters for the command. Commands that have no parameters (such as IOCTL_DEVICE_NO_COMMAND) will omit this parameter, and use the first form of the call.

The specific operation performed varies according to the `command` argument. The `command` argument also governs the use and interpretation of any additional arguments. The set of supported ioctl services is defined in the following sections.

Usage of all IOCTL calls is similar. Below is an example of a call using `IOCTL_DEVICE_READ_REGISTER` to read the contents of the board control register (BCR):

```
#include "opto32a_ioctl.h"

int ReadTest(int fd)
{
    device_register_params RegPar;
    int res;

    regdata.ulRegister = BOARD_STATUS_REG;
    regdata.ulValue = 0x0000; // to make sure it changes.
    res = ioctl(fd, (unsigned long)
                IOCTL_DEVICE_READ_REGISTER, &regdata);
    if (res < 0) {
        printf("%s: ioctl IOCTL_READ_REGISTER failed\n", argv[0]);
        }
    return (res);
```

2.8.1. IOCTL_DEVICE_NO_COMMAND

NO-OP call. IOCTL_DEVICE_NO_COMMAND is useful for verifying that the board has been opened properly.

Usage

`ioctl()` Argument	Description
request	IOCTL_DEVICE_NO_COMMAND

2.8.2. IOCTL_DEVICE_READ_REGISTER

This service reads the value of an OPTO32A register. This includes all GSC specific registers. Refer to `opto32a_ioctl.h` for a complete list of the accessible registers.

Usage

`ioctl()` Argument	Description
request	IOCTL_DEVICE_READ_REGISTER
arg	device_register_params*

2.8.3. IOCTL_DEVICE_WRITE_REGISTER

This service writes a value to an OPTO32A register. This includes only the GSC specific registers. All PCI and PLX PCI9080 feature set registers are not accessible. Refer to `opto32a_ioctl.h` for a complete list of the accessible registers.

Usage

`ioctl()` **Argument**	**Description**
`request`	`IOCTL_DEVICE_WRITE_REGISTER`
`arg`	`device_register_params*`

2.8.4. IOCTL_DEVICE_GET_DEVICE_TYPE

Returns a unique enumerated type for each board type supported. Allows the use of multiple varieties of General Standards boards in the same system. Possible return values are:

```
DEVICE_OPTO32A
```

Usage

`ioctl()` **Argument**	**Description**
`Request`	`IOCTL_DEVICE_GET_DEVICE_TYPE`
`Arg`	`unsigned long *`

2.8.5. IOCTL_DEVICE_MASTER_CLEAR

Activates the “master clear” bit in the board control register to clear all interrupt events.

Usage

`ioctl()` **Argument**	**Description**
`request`	`IOCTL_DEVICE_MASTER_CLEAR`
`arg`	`none`

2.8.6. IOCTL_DEVICE_GET_RECEIVED_DATA

Returns the contents of the received data register. See the `opto32a_ioctl.h` file for a list of possible return values.

Usage

`ioctl()` **Argument**	**Description**
`request`	`IOCTL_DEVICE_GET_RECEIVED_DATA`
`arg`	`unsigned long*`

2.8.7. IOCTL_DEVICE_SET_COS_REGISTER

Set the value of the Change of State (COS) register. See the `opto32a_ioctl.h` file for a list of possible values.

Usage

`ioctl()` **Argument**	**Description**
`request`	`IOCTL_DEVICE_SET_COS_REGISTER`
`arg`	`unsigned long*`

2.8.8. IOCTL_DEVICE_READ_EVENT_COUNT

Reads and returns the contents of the Event Count register. The contents of the register are not changed.

Usage

`ioctl()` **Argument**	**Description**
`request`	`IOCTL_DEVICE_READ_EVENT_COUNT`
`arg`	`unsigned long*`

2.8.9. IOCTL_DEVICE_SET_COS_IRQ

Selects which inputs will cause a change-of-state interrupt request. Setting this register does not actually enable the interrupt(s), rather the interrupt is enabled in the appropriate device_wait IOCTL. For valid bit-fields see the `opto32a_ioctl.h` file.

Usage

`ioctl()` **Argument**	**Description**
`request`	`IOCTL_DEVICE_SEST_COS_IRQ`
`arg`	`unsigned long*`

2.8.10. IOCTL_DEVICE_SET_COS_POLARITY

Set the polarity of the change-of-state interrupts. For valid bit-fields see the `opto32a_ioctl.h` file.

Usage

`ioctl()` **Argument**	**Description**
`request`	`IOCTL_DEVICE_SET_COS_POLARITY`
`arg`	`unsigned long*`

2.8.11. IOCTL_DEVICE_SET_DEVICE_CLOCK

Set the debounce clock divisor. See the hardware manual for a description of how to select the clock divisor. Range:

```
0-CDR_DIVISOR_MASK
```

Usage

`ioctl()` **Argument**	**Description**
`request`	`IOCTL_DEVICE_SET_DEVICE_CLOCK`
`arg`	`unsigned long*`

2.8.12. IOCTL_DEVICE_WRITE_DATA

Set the output data register. For valid bit-fields see the `opto32a_ioctl.h` file.

Usage

`ioctl()` **Argument**	**Description**
`request`	`IOCTL_DEVICE_WRITE_DATA`
`arg`	`unsigned long*`

2.8.13. IOCTL_DEVICE_WAIT_LO_COS

Tells the driver to wait for the low-byte change of state interrupt. The driver will sleep until the interrupt occurs, a cancel IOCTL is sent by the application or until a driver timeout; then return. The return value will indicate if the wait was successful.

Usage

`ioctl()` **Argument**	**Description**
`request`	`IOCTL_DEVICE_WAIT_LO_COS`
`arg`	`none`

2.8.14. IOCTL_DEVICE_WAIT_MID_COS

Tells the driver to wait for the mid-byte change of state interrupt. The driver will sleep until the interrupt occurs, a cancel IOCTL is sent by the application or until a driver timeout; then return. The return value will indicate if the wait was successful.

Usage

`ioctl()` **Argument**	**Description**
`request`	`IOCTL_DEVICE_WAIT_MID_COS`
`arg`	`none`

2.8.15. IOCTL_DEVICE_WAIT_HI_COS

Tells the driver to wait for the high-byte change of state interrupt. The driver will sleep until the interrupt occurs, a cancel IOCTL is sent by the application or until a driver timeout; then return. The return value will indicate if the wait was successful.

Usage

`ioctl()` **Argument**	**Description**
`request`	`IOCTL_DEVICE_WAIT_HI_COS`
`arg`	`none`

2.8.16. IOCTL_DEVICE_WAIT_EVENT_OVERFLOW

Tells the driver to wait for the event-overflow interrupt. The driver will sleep until the interrupt occurs, a cancel IOCTL is sent by the application or until a driver timeout; then return. The return value will indicate if the wait was successful.

Usage

`ioctl()` Argument	Description
`request`	`IOCTL_DEVICE_WAIT_EVENT_OVERFLOW`
`arg`	`unsigned long*`

2.8.17. IOCTL_DEVICE_SET_EVENT_TIMEOUT

Set the timeout for each type of wait event. Possible values for the event field are:

```
EVENT_BYTE_LO
EVENT_BYTE_MID
EVENT_BYTE_HI
EVENT_OVERFLOW
```

And possible values for the `timeout` field are integers in the range of 0x00-0xFFFFFFFF.

Usage

`ioctl()` Argument	Description
`request`	`IOCTL_DEVICE_SET_EVENT_TIMEOUT`
`arg`	`struct device_timeout_params*`

2.8.18. IOCTL_GET_DEVICE_ERROR

This call is used to retrieve the detailed error code for the most recent error. Not all error codes are used by, or are relevant to the driver. Possible return values are:

```
DEVICE_SUCCESS
DEVICE_INVALID_PARAMETER
DEVICE_INVALID_BUFFER_SIZE
DEVICE_PIO_TIMEOUT
DEVICE_DMA_TIMEOUT
DEVICE_IOCTL_TIMEOUT
DEVICE_OPERATION_CANCELLED
DEVICE_RESOURCE_ALLOCATION_ERROR
DEVICE_INVALID_REQUEST
DEVICE_AUTOCAL_FAILED
DEVICE_OVERFLOW
```

2.8.19. IOCTL_DEVICE_CANCEL

Cancels any pending wait IOCTLs and cleans up for driver exit.

Usage

`ioctl()` **Argument**	**Description**
`request`	`IOCTL_DEVICE_CANCEL`
`arg`	`unsigned long*`

2.8.20. IOCTL_DEVICE_READ_PLX_RUNTIME

This IOCTL is used to read the runtime registers of the PLX chip. The address to read is passed in 32-bit format (i.e. the address listed in the PLX data sheet divided by four). The runtime registers are defined in the file plx_regs.h. Arguments are passed in the device_register_params structure.

Usage

`ioctl()` **Argument**	**Description**
`request`	`IOCTL_DEVICE_READ_PLX_RUNTIME`
`arg`	`Struct device_register_params*`

Document History

Revision	**Description**
January 24, 2005	Initial draft.
July 15, 2005	Added support for reading the PLX runtime registers.