OPTO32

24 Optically Isolated Inputs, 8 Optically Isolated Outputs

All Form Factors
…-OPTO32/A/B/C/D

Linux Driver
User Manual

Manual Revision: June 20, 2023
Driver Release Version 8.7.104.47.0

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. Introduction

1.1. Purpose

The purpose of this document is to describe the interface to the OPTO32 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 OPTO32 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
COS	Change of State
CPCI	Compact PCI
DIO	Digital I/O
GSC	General Standards Corporation
PCI	Peripheral Component Interconnect
PCIe	PCI Express
PMC	PCI Mezzanine Card

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 OPTO32 installation directory or any of its subdirectories.
API Library	This is a library that provides application-level access to OPTO32 hardware.
Application	This is a user mode process, which runs in user space with user mode privileges.
D23	This is a reference that means “bit 23.”
Driver	This is the OPTO32 device driver, which runs in kernel space with kernel mode privileges.
Library	This is usually a general reference to the API Library.
OPTO32	This is used as a general reference to any board supported by this driver.

1.4. Software Overview

1.4.1. Basic Software Architecture

This section describes the general architecture for the basic components that comprise OPTO32 applications. The overall architecture is illustrated in Figure 1 below.

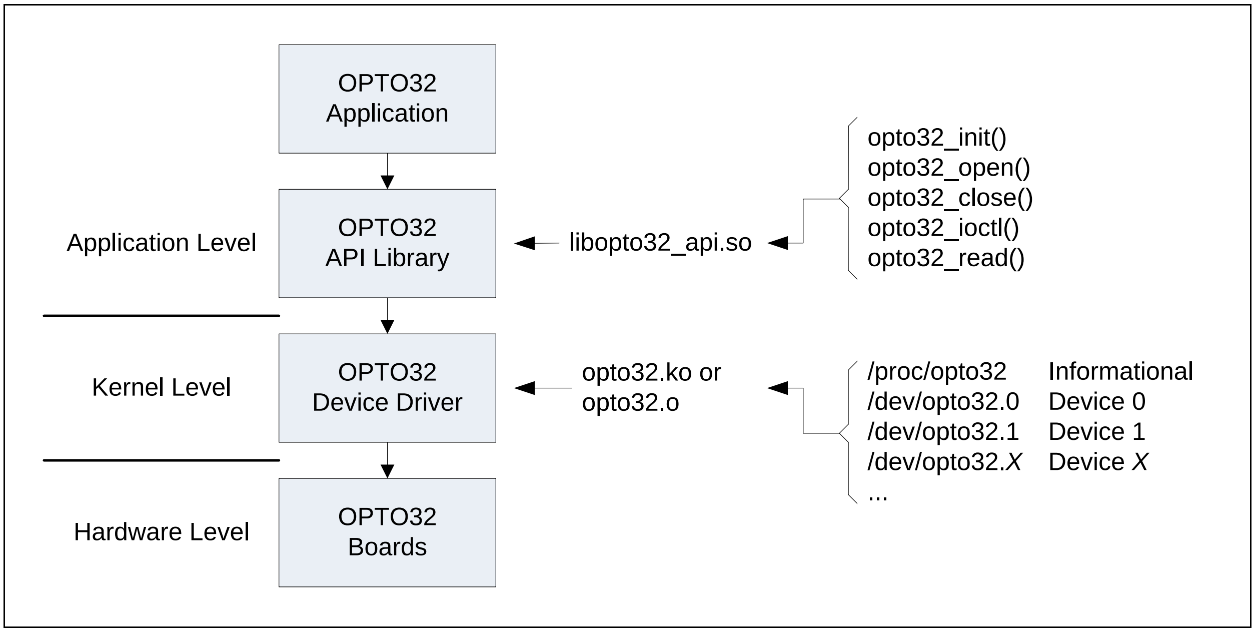

Figure 1 Basic architectural representation.

1.4.2. API Library

The primary means of accessing OPTO32 boards is via the OPTO32 API Library. This library forms a layer between the application and the driver. Additional information is given in section 4 (page 16). With the library, applications are able to open and close a device and, while open, perform I/O control and read and write operations.

1.4.3. Device Driver

The device driver is the host software that provides a means of communicating directly with OPTO32 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 OPTO32 is a high performance optically isolated I/O board with Change of State (COS) detection on all inputs. The board includes 24 optically isolated inputs and eight optically isolated outs. All inputs are controlled by a global debounce timer. The debounce period consists of three sampling intervals, where the interval is configurable in 100ns increments from 200ns to just under 215 seconds. The denounced input can be read at any time. Also, each COS input can be independently configured to generate an interrupt on either a rising or falling state transition. The D23 input has the added capability of generating an interrupt after receipt of from 1 to 64K low-to-high state changes. The eight outputs include four with normal output capability and four with high output capability.

1.6. Reference Material

The following reference material may be of particular benefit in using the OPTO32. The specifications provide the information necessary for an in depth understanding of the specialized features implemented on this board.

- The *OPTO32 API Library Reference Manual* from General Standards Corporation.
- The applicable *OPTO32 User Manual* from General Standards Corporation.

- The *PCI9056 PCI Bus Master Interface Chip* data handbook from PLX Technology, Inc. *
- The *PCI9060ES PCI Bus Master Interface Chip* data handbook from PLX Technology, Inc. *

* PLX data books are available from PLX at the following location.

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 6.x, 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
6.0.7	Red Hat Fedora Core 37
5.17.5	Red Hat Fedora Core 36
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 is designed for SMP support, but has not undergone SMP specific testing.

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/opto32` file will be "`no`".

2.2. The /proc/ File System

While the driver is running, the text file `/proc/opto32` can be read to obtain information about the driver and the boards it detects. 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: 8.7.104.47
32-bit support: yes
boards: 1
models: OPTO32
```

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 basic model number for each board the driver detected. The model numbers are listed in the same order that the boards are accessed via the API Library's open function. For this driver all model numbers should be `OPTO32`.

2.3. File List

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

File	Description
`opto32.linux.tar.gz`	This archive contains the driver, the API Library and all related files.
`opto32_api_rm.pdf`	This is a PDF version of the *OPTO32 API Library Reference Manual*.
`opto32_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	Description
`opto32/`	This is the driver root directory. It contains the documentation, the Overall Make Script (section 2.7, page 11) and the below listed subdirectories.
…`/api/`	This directory contains the API Library source files (section 4, page 16).
…`/docsrc/`	This directory contains the source files for the code samples given in the reference manual (section 6, page 22).
…`/driver/`	This directory contains the device driver source files (section 5, page 18).
…`/include/`	This directory contains the header 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 25).
`…/utils/`	This directory contains the source files for the utility libraries used by the sample applications (section 7, page 23).

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 `opto32.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 `opto32` in the current directory, and then copies all of the archive's files into this new directory.

```
tar –xzvf opto32.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 (page 21).

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 opto32.linux.tar.gz opto32
```

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

```
rm -f /dev/opto32.*
```

5. If the automatic startup procedure was adopted (section 5.3.2, page 19), then edit the system startup script `rc.local` and remove the line that invokes the OPTO32'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 and to load the driver.

NOTE: The following steps may require elevated privileges.

1. Change to the driver root directory (`…/opto32/`).

2. Remove existing build targets using the below command. This does not unload the driver.

```
./make_all clean
```

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

```
./make_all
```

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/libopto32_api.so`
Defined and Not Empty	`==== Linking: ../lib/libopto32_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/opto32_utils.a`
Defined and Not Empty	`==== Linking: ../lib/opto32_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 OPTO32 based applications. For additional information refer to the *OPTO32 API Library Reference Manual*.

3.1. Main Header File

Throughout the remainder of this document references are made to various header files included as part of the OPTO32 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 OPTO32 specific header files. Therefore, sources may include only this one OPTO32 header and make files may reference only this one OPTO32 include directory.

Description	File	Location
Header File	`opto32_main.h`	`…/include/`

3.2. Main Library File

Throughout the remainder of this document references are made to various statically linkable libraries included with the driver. 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 static libraries included with the driver. Therefore, make files may reference only this one OPTO32 static library and only this one OPTO32 library directory.

Description	File	Location
Static Libraries	`opto32_main.a`	`…/lib/`
	`opto32_multi.a`	

NOTE: For applications using the OPTO32 and no other GSC devices, link the `opto32_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.

NOTE: The OPTO32 API Library is implemented as a shared library and is thus not linked with the OPTO32 Main Library. The API Library must be linked with applications by adding the linker argument `–lopto32_api` to the linker command line.

3.2.1. Build

The main library is built via the Overall Make Script (section 2.7, page 11). 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, as well as the API Library shared object file, 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 OPTO32 API Library is the software interface between user applications and the OPTO32 device driver. The interface is accessed by including the header file `opto32_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.

Description	File	Location
Source Files	`*.c`, `*.h`, `makefile` …	…`/api/`
Header File	`opto32_api.h`	…`/include/`
Library File	`libopto32_api.so`	…`/lib/` `/usr/lib/` †

† The shared object 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 11), but can be built separately following the below steps.

NOTE: The following 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. This step copies the API Library file to `/usr/lib/`.

   ```
   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. Also, edit the include file search path to locate the header file in the below listed directory. At link time the Library's shared object file is linked via the linker command line. This can be done by naming the .so file explicitly or by adding the below linker command line argument. At run time the library is found in the directory `/usr/lib/`. (The shared object file is automatically copied to `/usr/lib/` when it is built.)

Description	File	Location	Linker Argument
Header File	`opto32_api.h`	…`/include/`	
Shared Object Library	`libopto32_api.so`	…`/lib/`	
		`/usr/lib/`	`-lopto32_api`

4.4. Macros

For detailed macro information refer to this same section number in the *OPTO32 API Library Reference Manual*.

4.5. Data Types

For detailed data type information refer to this same section number in the *OPTO32 API Library Reference Manual*.

4.6. Functions

For detailed function information refer to this same section number in the *OPTO32 API Library Reference Manual*.

4.7. IOCTL Services

For detailed IOCTL information refer to this same section number in the *OPTO32 API Library Reference Manual*.

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.

Description	**Files**	**Location**
Source Files	`*.c`, `*.h`, `Makefile` …	…`/driver/`
Header File	`opto32.h`	
Driver File	`opto32.ko` † `opto32.o` ‡	

† This is for kernel versions 2.6 and later.
‡ This is for kernel versions 2.4 are earlier.

5.2. Build

NOTE: Building the driver requires installation of the kernel headers and possibly other packages.

The device driver 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 driver and its sources are installed (…`/driver/`).

2. Remove existing build targets by issuing 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 load the device driver and create fresh device nodes. This is accomplished by unloading the current driver, if loaded, and then loading the accompanying driver executable. In addition, the script 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 corresponds 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 is 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 booted.

   NOTE: The OPTO32 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 `opto32` 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/opto32.*
   ```

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/opto32/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 your 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/opto32` 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/opto32
   ```

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/opto32` 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 opto32
   ```

2. Verify that the driver module has been unloaded by issuing the below command. The module name `opto32` should not be in the listed output.

   ```
   lsmod
   ```

6. Document Source Code Examples

This is a library of the source code included in the *OPTO32 API Library Reference Manual*. For additional information refer to the *OPTO32 API Library Reference Manual*.

6.1. Files

The library files are summarized in the table below.

Description	**Files**	**Location**
Source Files	`*.c`, `*.h`, `makefile` …	…`/docsrc/`
Header File	`opto32_dsl.h`	…`/include/`
Library File	`opto32_dsl.a`	…`/lib/`

6.2. 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 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 14).

6.3. Library Use

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

7. Utilities Source Code

The driver archive includes a body of utility source code designed to aid in the understanding and use of the API calls and the IOCTL services. The utility services provide wrappers, mostly visual, around the respective services. Utility sources are also included for device independent and common, general-purpose services. The aim of all the visual wrappers is to facilitate structured console output for the sample applications. The utility services are used extensively by 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. For additional information refer to the *OPTO32 API Library Reference Manual*.

7.1. Files

The utility files are summarized in the table below.

Description	**Files**	**Location**
Source Files	`*.c`, `*.h`, `makefile` …	…`/utils/`
Header File	`opto32_utils.h`	…`/include/`

Library Files	`opto32_utils.a`
	`gsc_utils.a`
	`os_utils.a`
	`plx_utils.a`
		…`/lib/`

7.2. Build

The libraries are 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 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 3.2.1, page 14).

7.3. Library Use

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

8. Operating Information

For operating information refer to this same section number in the *OPTO32 API Library Reference Manual*.

9. Sample Applications

The driver archive includes a variety of sample and test applications located under the `samples` subdirectory. 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 11), but each may be built individually by changing to its respective directory and issuing the commands “`make clean`” and “`make`”. The initial output from each application includes information on its supported command line arguments. The following gives a brief overview of each application.

> **NOTE**: None of the sample application are specifically written to support simultaneous execution. The applications may function satisfactorily when multiple instances are run simultaneously on the same serial channel or board, but they may not.

9.1. din - Digital Input - …/din/

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. 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.

9.4. led - LED Exerciser - …/led/

This application exercises the board LEDs.

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. 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**
June 20, 2023	Updated to release version 8.7.104.47.0. Updated the kernel support table. Minor editorial changes.
October 7, 2022	Updated to release version 8.6.101.43.0.
July 5, 2022	Updated to release version 8.5.100.41.0. Minor editorial changes.
June 30, 2022	Updated to release version 8.5.99.41.0. Expanded automatic startup information. Updated the kernel support table. Added section on environment variables. Removed all references to the SDK.
March 25, 2021	Updated to release version 8.4.93.36.0. Updated the kernel support table. Added SDK discontinuation notice. Various editorial changes.
May 22, 2020	Updated to release version 8.4.91.31.0. Expanded automatic startup information.
August 9, 2019	Updated to release version 8.3.87.28.1. Various editorial updates.
July 17, 2019	Updated to release version 8.3.87.28.0. Updated the kernel support table. Minor editorial changes. Added a licensing subsection. Document reorganization.
September 6, 2018	Updated to release version 8.2.80.26.0. Updated the inside cover page. Document reorganization. Split content into this Linux user manual and an *OPTO32 API Library Reference Manual*.
June 13, 2018	Updated to version 6.1.69.18.1, which consists only of an updated driver user manual. Updated the inside cover page. The kernel support table has been updated, but this alteration is for an upcoming driver release.
December 16, 2016	Updated to version 6.1.69.18.0. Removed the `built` field from the `/proc` file. Updated the kernel support table. Organized sample applications alphabetically. Updated the usage of the Wait Event `timeout_ms` field. 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.
September 16, 2015	Updated to version 8.0.60.8.0. Updated the device node name to include a period before the device index. Removed double underscore that prefaced various data types.
February 28, 2014	Updated to version 7.7.52.0. Updated the kernel support table.
January 9, 2014	Updated to version 7.6.51.0. Updated the kernel support table.
November 8, 2013	Updated to version 7.6.48.0.
September 10, 2013	Updated to version 7.5.47.0. Ported the SDK to this driver. Added all previous SDK components to this release. Removed the `tftest` sample application.
July 7, 2013	Updated to version 7.4.45.0. Updated the kernel support table.
July 24, 2012	Updated to version 7.4.39.0. Updated the kernel support table.
December 21, 2011	Updated to version 7.3.34.0.
November 10, 2011	Updated to version 7.2.32.0.
May 29, 2011	Updated to version 7.1.25.0.
January 17, 2011	Initial release, version 7.0.22.0. This release replaces the SDK and all previous driver releases.