A Simple I/O Library for Embedded Linux Systems

http://git.munts.com/libsimpleio

By Philip Munts
President, Munts AM Corp

Contents

Revision History

Revision 1, 9 August 2017	Initial draft.
Revision 2, 11 August 2017	Added revision history. Added installation instructions. Added note that **`libsimpleio`** can be used with desktop Linux. Added note that calling **`EVENT_wait()`** with zero timeout can be used for polling without blocking.
Revision 3, 15 November 2017	Added **`libadc`**, which provides services for Linux Industrial I/O Subsystem ADC (Analog to Digital Converter) inputs. Upgraded from Debian Jessie to Stretch.
Revision 4, 12 May 2018	Switched from the old deprecated GPIO sysfs API to the new GPIO descriptor API.
Revision 5, 14 May 2018	Header files are now installed to **`/usr/local/include/libsimpleio`**.
Revision 6, 4 February 2019	Added **`libdac`**, which provides services for Linux Industrial I/O Subsystem DAC (Digital to Analog Converter) outputs. Renamed the repository for MuntsOS Embedded Linux Framework from http://git.munts.com/arm-linux-mcu to http://git.munts.com/muntsos.
Revision 7, 7 February 2019	Updated stale links and commands.
Revision 8, 3 January 2020	Changed **`debian9`** to **`debian10`**.
Revision 9, 8 January 2020	Added **`LINUX_errno()`**.
Revision 10, 17 February 2020	Mention Remote I/O Protocol devices. Moved **`git`** repositories to GitHub.
Revision 11, 30 April 2020	Added **`popen()`** and **`pclose()`** wrappers.
Revision 12, 17 September 2020	Updated the installation procedures.
Revision 13, 5 December 2020	Renamed **`HIDRAW_open()`** to **`HIDRAW_open1()`**. Renamed **`HIDRAW_open_id()`** to **`HIDRAW_open2()`**. Added **`HIDRAW_open3()`**.

Introduction

Rationale

libsimpleio is an attempt to regularize the mish-mash of API styles that Linux presents for I/O device access. The support for I/O devices in Linux has evolved over time such that there are many different and incompatible API styles. For example, an application program must use **ioctl()** to access SPI (Serial Peripheral Interconnect) devices, **tcsetattr()** and other functions defined in **termios.h** to access serial port devices, and Berkeley sockets library functions to access network devices.

libsimpleio exports C functions with a common and highly regular calling sequence that encapsulate and hide the underlying Linux system call services. These C functions are callable from Ada, C#, Java, and Free Pascal application programs, using the native or external library binding facility each language provides.

Although primarily intended for dedicated embedded Linux systems (such as ***MuntsOS Embedded Linux***), **libsimpleio** is also usable on mainstream desktop Linux systems such as Debian or Ubuntu.

Copyright and Licensing

All original works in **libsimpleio** are copyrighted and licensed according the following 1-clause BSD source code license:

Online Resources

The web site and **`git`** source code repository for **`libsimpleio`** is available at:

https://github.com/pmunts/libsimpleio

The **`man`** pages specifiying the API (Application Program Interface) for **`libsimpleio`** are available for browsing at:

http://git.munts.com/libsimpleio/doc/libsimpleio.html

Prebuilt **`libsimpleio`** packages for Debian Stretch and compatible Linux operating systems are available for download at:

http://repo.munts.com/debian10

The web site and **`git`** source code repository for the ***MuntsOS Embedded Linux Framework*** is available at:

https://github.com/pmunts/muntsos

Installation

From the Munts Technologies Debian Package Repository

The easiest way to use **libsimpleio** on Debian Linux and its derivatives is to add the ***Munts Technologies Debian Package Repository*** to your system, with the following commands:

```
wget http://repo.munts.com/debian10/PublicKey.txt
sudo apt-key add PublicKey.txt
```

For 64-Bit x86 Debian Linux:

```
wget http://repo.munts.com/debian10/amd64/munts-repo.deb
sudo dpkg -i munts-repo.deb
```

For 32-Bit ARMv6 Raspberry Pi OS (formerly known as Raspbian):

```
wget http://repo.munts.com/debian10/raspberrypi/munts-repo.deb
sudo dpkg -i munts-repo.deb
```

Then you can install the native **libsimpleio** package with these commands:

```
apt-get update
apt-get install munts-libsimpleio
```

The package installs files into several directories under the **/usr/local** directory tree:

```
/usr/local/include/libsimpleio
/usr/local/lib
/usr/local/share/libsimpleio
/usr/local/share/man
```

There are also cross-compiled packages for ***MuntsOS Embedded Linux*** available in the repository. You can install them with the following commands:

```
apt-get install gcc-aarch64-linux-gnu-muntsos-linaro-libsimpleio
apt-get install gcc-arm-linux-gnueabihf-muntsos-BeagleBone-linaro-libsimpleio
apt-get install gcc-arm-linux-gnueabihf-muntsos-RaspberryPi-linaro-libsimpleio
apt-get install gcc-arm-linux-gnueabihf-muntsos-RaspberryPi2-linaro-libsimpleio
```

From Source Checkout

On an **dpkg** based Linux distribution (Debian, Ubuntu, *etc.*), **libsimpleio** can be built and installed from a source checkout with the following commands:

```
git clone https://github.com/pmunts/libsimpleio.git
cd libsimpleio
make
sudo dpkg -i *.deb
```

On other types of Linux distribution (Red Hat, Suse, *etc.*), **libsimpleio** can be built and installed from a source checkout with the following commands:

```
git clone https://github.com/pmunts/libsimpleio.git
cd libsimpleio
make
sudo make install
sudo make install_udev_rules [UDEVRULESDIR=<some other directory>]
```

The default location for udev rules is **/etc/udev/rules.d**. If your Linux distribution has them somewhere else, you can override the default location with the **UDEVRULESDIR** argument to the **make** command.

Coding Conventions

Naming

Each C function exported by **libsimpleio** is named according to the following convention:

<SUBSYSTEM>_<operation>()

where the prefix **<SUBSYSTEM>** indicates a particular I/O subsystem (I^2C, SPI, watchdog timer, etc.) and the suffix **<operation>** indicates a particular I/O operation (open, close, etc.). The subsystem prefix is always spelled in all capital letters and the operation suffix is always spelled in all lower case letters. *Examples:*

- **GPIO_configure()**
- **I2C_transaction()**
- **WATCHDOG_open()**

All C function declarations are bracketed between **_BEGIN_STD_C** and **_END_STD_C** macros (defined in **cplusplus.h**) to prevent name mangling when a header file is included by a C++ source file.

Constants exported by **libsimpleio**, whether defined with **#define** or with t**ypedef enum**, are always spelled with all capital letters. *Examples:*

- **GPIO_DIRECTION_INPUT**
- **CMD_SPI_OPEN**

*Note: The language bindings do **not** necessarily adhere to the above naming conventions.*

Argument Passing

All C functions exported by **`libsimpleio`** shall be proper procedures without any return value.

All numeric and enumeration type arguments shall be defined as 32-bit signed integers (**`int32_t`**), unless there is compelling reason otherwise. Each numeric argument shall be passed by value (**`int32_t x`**). If a value is to be returned, a numeric or enumeration type argument shall be passed by reference (**`int32_t *x`**). *Example:*

```
void GPIO_close(int32_t fd, int32_t *error);
```

All string arguments shall be **`NUL`** terminated C strings. Each string parameter shall be defined with: **`const char *`**, or, if a value is to be returned: **`char *`**. *Example:*

```
LINUX_drop_privileges(const char *username, int32_t *error);
```

Language Bindings

Each language binding (available for Ada, C#, Java, and Pascal) shall consist only of type, constant and function declarations that map each C type, constant or function declaration into the target programming language.

For a particular I/O subsystem, all of the language bindings (*e.g.* **`libgpio.h`**, **`libgpio.ads`**, **`libgpio.cs`**, **`libgpio.java`**, and **`libgpio.pas`**) shall be functionally identical.

I/O Subsystem Overviews

Please refer to the **man** pages (online at the links below, from the **man** command, or in the appendix to this document) for the exact API specification for each subsystem.

libadc

http://git.munts.com/libsimpleio/doc/libadc.html

This library provides wrapper functions to read from Linux Industrial I/O Subsystem ADC (Analog to Digital) Converters. Use **ADC_open()** to open an analog input device. Use **ADC_read()** to sample an analog input.

*Note: **ADC_read()** returns an analog sample as a right justified 32-bit integer. The meaning of this integer (i.e. the ADC resolution) is system dependent.*

libdac

http://git.munts.com/libsimpleio/doc/libdac.html

This library provides wrapper functions to write to Linux Industrial I/O Subsystem DAC (Digital to Analog) Converters. Use **DAC_open()** to open an analog input device. Use **DAC_write()** to write to an analog output.

*Note: **DAC_write()** accepts an analog sample as a right justified 32-bit integer. The meaning of this integer (i.e. the DAC resolution) is system dependent.*

libevent

http://git.munts.com/libsimpleio/doc/libevent.html

This library provides wrapper functions for the Linux **epoll** I/O event notification system call functions. Use **EVENT_open()** to create an event handler. Use **EVENT_register()** to register a file descriptor for events. Use **EVENT_wait()** to suspend the process until an event occurs.

*Note: **EVENT_wait()** suspends the entire calling process (all threads).*

*Note: Passing a **timeoutms** value of zero to **EVENT_wait()** causes it to return immediately, and can be used to poll the availablity of data without blocking at all.*

libgpio

http://git.munts.com/libsimpleio/doc/libgpio.html

This library provides wrapper functions for the Linux GPIO pin services, using the new GPIO descriptor API. Each GPIO pin is identified by a hardware dependent pair of chip (subsystem) and line (pin) numbers. Use **`GPIO_line_open()`** to initialize and open a GPIO pin, **`GPIO_line_read()`** to read from a GPIO pin, and **`GPIO_line_write()`** to write to a GPIO pin.

Use **`GPIO_line_event()`** to wait for input edge events on a GPIO pin that has been configured as an interrupt input.

libhidraw

http://git.munts.com/libsimpleio/doc/libhidraw.html

This library provides wrapper functions for the Linux raw HID device **`ioctl()`** services. The Linux kernel raw HID subsystem creates a device node of the form **`/dev/hidrawN`** for each raw HID device detected. Use **`HIDRAW_open1()`** to open a raw HID device node by name. Use **`HIDRAW_open2()`** to open the first raw HID device to match the given vendor and product identifiers. Use **`HIDRAW_open3()`** to open the HID device matching the given vendor and product identifiers and serial number. Use **`HIDRAW_send()`** to send a 64-byte message (*aka* HID report) to the raw HID device. Use **`HIDRAW_receive()`** to obtain a 64-byte message (*aka* HID report) from the raw HID device.

*Note: The message size parameter passed to **`HIDRAW_send()`** and **`HIDRAW_receive()`** will typically be 64 bytes, or 65 bytes if the raw HID device uses the first byte of the message for the report number.*

*Note: **`HIDRAW_send()`** and **`HIDRAW_receive()`** are blocking functions. If you need to wait for the raw HID device without blocking, you can register its file descriptor with **`EVENT_register()`** and wait for something to happen with **`EVENT_wait()`**.*

libi2c

http://git.munts.com/libsimpleio/doc/libi2c.html

This library provides wrapper functions for the Linux I^{2}C device **`ioctl()`** services. The Linux kernel I^{2}C subsystem creates a device node of the form **`/dev/i2c-N`** for each I^{2}C bus controller detected. Each I^{2}C bus may have one or more slave devices attached to it. Use **`I2C_open()`** to open an I^{2}C bus controller device node by name. Use **`I2C_transaction()`** to transmit a command and/or receive a response from a single I^{2}C device.

liblinux

http://git.munts.com/libsimpleio/doc/liblinux.html

This library provides wrapper functions for certain Linux system calls. Use **`LINUX_detach()`** to switch a running program from foreground to background execution. Use **`LINUX_drop_privileges()`** change a running program's user (*e.g.* from **`root`** to **`nobody`**). Use **`LINUX_openlog()`** to initialize a connection to the **`syslog`** facility. Use **`LINUX_syslog()`** to post a message to the **`syslog`** facility. Use **`LINUX_errno()`** to retrieve the value of **`errno`**. Use **`LINUX_strerror()`** to retrieve the error message associated with an **`errno`** error number. Use **`LINUX_poll()`** to wait for I/O on a file descriptor. Use **`LINUX_usleep()`** to sleep for some number of microseconds. Use **`LINUX_command()`** to execute a shell command string. Use **`LINUX_popen_read()`** to open a pipe to read standard output from another program. Use **`LINUX_popen_write()`** to open a pipe to write standard input to another program. Use **`LINUX_pclose()`** to close a pipe to or from another program.

For the C or C++ programming languages, these wrapper functions offer no particular benefit over the regular system call functions provided by **`glibc`**.

liblinx

http://git.munts.com/libsimpleio/doc/liblinx.html
https://www.labviewmakerhub.com/doku.php?id=learn:libraries:linx:spec:start

This library provides functions for sending and receiving messages between a client program and a Labview LINX remote I/O device. To develop a LINX ***client*** program, use **`LINX_transmit_command()`** and **`LINX_receive_response()`**. To develop a LINX ***server*** program, use **`LINX_receive_command()`** and **`LINX_transmit_response()`**. All four of these functions require an open byte stream file descriptor (*e.g.* from **`SERIAL_open()`** or **`TCP4_connect()`**) as the first parameter.

Each of the receive functions returns after accepting one byte from the byte stream. A value of zero in the **`error`** parameter indicates that the received byte has completed a frame and **`EAGAIN`** indicates otherwise. Each of the receive functions must be called successively with the same arguments until the frame has been completed.

*Note: The receive functions block until a byte is available from the underlying byte stream. If you need to wait without blocking, you can register the byte stream file descriptor with **`EVENT_register()`** and wait for something to happen with **`EVENT_wait()`**.*

libpwm

http://git.munts.com/libsimpleio/doc/libpwm.html

This library provides functions for configuring and writing to PWM (Pulse Width Modulated) output devices. The Linux kernel PWM subsystem populates **`sysfs`** entries for each PWM output configured. PWM outputs are identified by chip and channel numbers. Use **`PWM_configure()`** to configure a single PWM output. Use **`PWM_open()`** to open a single PWM device node. Use **`PWM_write()`** to set the PWM output duty cycle.

Note: Many PWM controllers require the same PWM pulse frequency for all channels. Therefore, configuring different pulse period values for different channels within the same PWM controller may result in incorrect operation.

libserial

http://git.munts.com/libsimpleio/doc/libserial.html

This library provides wrapper functions for the Linux serial port **`termios`** services. The Linux kernel serial port subsystem creates a device node of the form **`/dev/ttyXXXX`** for each serial port device detected. Use **`SERIAL_open()`** to configure and open a serial port device by name. Use **`SERIAL_send()`** to send data to a serial port device. Use **`SERIAL_receive()`** to receive data from a serial port device.

*Note: The file descriptor returned by **`SERIAL_open()`** may be passed to **`STREAM_send()`** and **`STREAM_receive()`** as described below.*

*Note: **`SERIAL_send()`** and **`SERIAL_receive()`** are blocking functions. If you need to wait for the serial port device without blocking, you can register its file descriptor with **`EVENT_register()`** and wait for something to happen with **`EVENT_wait()`**.*

libspi

http://git.munts.com/libsimpleio/doc/libspi.html

This library provides wrapper functions for the Linux SPI device **`ioctl()`** services. The Linux kernel SPI subsystem creates a device node of the form **`/dev/spidev-X.Y`** for each SPI slave device detected. Use **`SPI_open()`** to open an SPI slave device node by name. Use **`SPI_transaction()`** to transmit a command and/or receive a response from a single SPI slave device.

*Note: Some hardware platforms may not implement hardware controlled slave select output signals. A GPIO pin file descriptor obtained with **`GPIO_open()`** may be passed to **`SPI_transaction()`** to request software controlled slave select.*

libstream

http://git.munts.com/libsimpleio/doc/libstream.html
http://git.munts.com/libsimpleio/doc/StreamFramingProtocol.pdf

This library provides functions for encoding and decoding byte stream data into frames as specified in the Stream Framing Protocol. A common use case for this protocol is to communicate with a microcontroller via a serial port. **`Use STREAM_encode_frame()`** to encode a frame for transmission. Use **`STREAM_decode_frame()`** to decoded a received frame. Use **`STREAM_send_frame()`** to transmit a frame via a byte stream indicated by a Linux file descriptor. Use **`STREAM_receive_frame()`** to receive a frame via a byte stream indicated by a Linux file descriptor.

`STREAM_receive_frame()` returns after accepting one byte from the byte stream. It will return zero in the **`error`** parameter if that byte completes a frame and **`EAGAIN`** otherwise. **`STREAM_receive_frame()`** must be called successively with the same arguments until the frame has been completed.

*Note: **`STREAM_receive_frame()`** blocks until a byte is available from the underlying byte stream. If you need to wait without blocking, you can register the byte stream file descriptor with **`EVENT_register()`** and wait for something to happen with **`EVENT_wait()`**.*

libipv4

http://git.munts.com/libsimpleio/doc/libipv4.html

This library provides wrapper functions for the Linux IPv4 socket services. Use **`IPV4_resolve()`** to resolve a host name to a 32-bit integer IPv4 address. Use **`IPV4_ntoa()`** to convert a 32-bit integer IPv4 address to a string, in dotted octet notation (*e.g.* **`1.2.3.4`**). Use **`TCP4_connect()`** to connect to a TCP server. Use **`TCP4_accept()`** or **`TCP4_server()`** to implement a TCP server. Use **`TCP4_send()`** to send data and **`TCP4_receive()`** to receive data.

*Note: The file descriptor returned by **`TCP4_connect()`**, **`TCP4_accept()`**, or **`TCP4_server()`** may be passed to **`STREAM_send()`** and **`STREAM_receive()`** as described above.*

*Note: **`TCP4_send()`** and **`TCP4_receive()`** are blocking functions. If you need to wait without blocking, you can register the file descriptor with **`EVENT_register()`** and wait for something to happen with **`EVENT_wait()`**.*

libwatchdog

http://git.munts.com/libsimpleio/doc/libwatchdog.html

This library provides functions for configuring and resetting watch dog timer devices. The Linux kernel watchdog timer subsystem creates a device node of the form **`/dev/watchdogN`** for each watchdog timer. The default watchdog timer is **`/dev/watchdog`**. Use **`WATCHDOG_open()`** to open a watchdog timer device node by name. Use **`WATCHDOG_get_timeout()`** to query the current period in seconds, and **`WATCHDOG_set_timeout()`** to change the period. Use **`WATCHDOG_kick()`** to reset the watchdog timer.

Man Pages

NAME

libsimpleio −− Linux Simple I/O Library

DESCRIPTION

libsimpleio is an attempt to encapsulate (as much as possible) the ugliness of Linux I/O device access. It provides services for reading and/or writing the following types of Linux I/O devices:

* Industrial I/O Subsystem A/D (Analog to Digital Converter) Devices
* Industrial I/O Subsystem D/A (Digital to Analog Converter) Devices
* GPIO (General Purpose Input/Output) Pins
* Raw HID (Human Interface Device) Devices
* I^2C (Inter-Integrated Circuit) Bus Devices
* LabView LINX Remote I/O Devices
* PWM (Pulse Width Modulated) Output Devices
* Remote I/O Protocol Devices
* Serial Ports
* SPI (Serial Peripheral Interface) Bus Devices
* Stream Framing Protocol Devices
* TCP and UDP over IPv4 Network Devices
* Watchdog Timer Devices

Although **libsimpleio** was originally intended for Linux microcomputers such as the Raspberry Pi, it can also be useful on larger desktop Linux systems.

SEE ALSO

libadc(2), **libdac**(2), **libevent**(2), **libgpio**(2), **libhidraw**(2), **libi2c**(2), **libipv4**(2), **liblinux**(2), **liblinx**(2), **libpwm**(2), **libserial**(2), **libspi**(2), **libstream**(2), **libwatchdog**(2)

http://git.munts.com/libsimpleio/doc/RemoteIOProtocol.pdf
http://git.munts.com/libsimpleio/doc/StreamFramingProtocol.pdf
http://git.munts.com/libsimpleio/doc/UserManual.pdf

AUTHOR

Philip Munts, President, Munts AM Corp dba Munts Technologies

NAME

libadc −− Linux Simple I/O Library: A/D Input Module

SYNOPSIS

#include <libsimpleio/libadc.h>

void ADC_get_name(int32_t *chip*, **char ***name*, **int32_t** *namesize*,
int32_t ***error*);

void ADC_open(int32_t *chip*, **int32_t** *channel*, **int32_t *** *fd*,
int32_t ***error*);

void ADC_close(int32_t *fd*, **int32_t ***error*);**

void ADC_read(int32_t *fd*, **int32_t ***sample*, **int32_t ***error*);**

Link with **-lsimpleio**.

DESCRIPTION

All functions return either **0** (upon success) or an **errno** value (upon failure) in **error*.

ADC_get_name() fetches an information string for an A/D device. The chip number must be passed in *chip*. The destination buffer address must be passed in **name* and the destination buffer size must be passed in *namesize*. The minimum size for the destination buffer is 16 bytes.

ADC_open() opens an A/D input channel device. The A/D chip number must be passed in the *chip* parameter and the A/D input channel number must be passed in the *channel* parameter. Upon success, a file descriptor for the A/D input channel device is returned in **fd*.

ADC_close() closes a previously opened A/D input channel device.

ADC_read() reads a sample from an A/D input channel device. The file descriptor for an open A/D input channel device must be passed in the *fd* parameter. The analog sample data will be returned in the **sample* parameter.

SEE ALSO

libsimpleio(2), **libdac**(2), **libevent**(2), **libgpio**(2), **libhidraw**(2),
libi2c(2), **libipv4**(2), **liblinux**(2), **liblinx**(2), **libpwm**(2),
libserial(2), **libspi**(2), **libstream**(2), **libwatchdog**(2)

https://wiki.analog.com/software/linux/docs/iio/iio

AUTHOR

Philip Munts, President, Munts AM Corp dba Munts Technologies

NAME

libdac −− Linux Simple I/O Library: D/A Input Module

SYNOPSIS

#include <libsimpleio/libdac.h>

void DAC_get_name(int32_t *chip***, char ****name***, int32_t** *namesize***,**
int32_t ****error*);**

void DAC_open(int32_t *chip***, int32_t** *channel***, int32_t ****fd***,**
int32_t ****error*);**

void DAC_close(int32_t *fd***, int32_t ****error***);**

void DAC_write(int32_t *fd***, int32_t** *sample***, int32_t ****error***);**

Link with **-lsimpleio**.

DESCRIPTION

All functions return either **0** (upon success) or an **errno** value (upon failure) in **error*.

DAC_get_name() fetches an information string for a D/A device. The chip number must be passed in *chip*. The destination buffer address must be passed in **name* and the destination buffer size must be passed in *namesize*. The minimum size for the destination buffer is 16 bytes.

DAC_open() opens a D/A output channel device. The D/A chip number must be passed in the *chip* parameter and the D/A output channel number must be passed in the *channel* parameter. Upon success, a file descriptor for the D/A output channel device is returned in **fd*.

DAC_close() closes a previously opened D/A output channel device.

DAC_write() writes a sample to a D/A output channel device. The file descriptor for an open D/A output channel device must be passed in the *fd* parameter. The analog sample data must be passed in the *sample* parameter.

SEE ALSO

libsimpleio(2), **libadc**(2), **libevent**(2), **libgpio**(2), **libhidraw**(2),
libi2c(2), **libipv4**(2), **liblinux**(2), **liblinx**(2), **libpwm**(2),
libserial(2), **libspi**(2), **libstream**(2), **libwatchdog**(2)

https://wiki.analog.com/software/linux/docs/iio/iio

AUTHOR

Philip Munts, President, Munts AM Corp dba Munts Technologies

NAME

libevent −− Linux Simple I/O Library: Event Notification Module

SYNOPSIS

#include <libsimpleio/libevent.h>

void EVENT_open(int32_t *_epfd_, int32_t *_error_);

void EVENT_close(int32_t *epfd*, **int32_t *_error_);**

void EVENT_register_fd(int32_t *epfd*, **int32_t** *fd*, **int32_t** *events*,
int32_t *handle*, **int32_t *_error_);**

void EVENT_modify_fd(int32_t *epfd*, **int32_t** *fd*, **int32_t** *events*,
int32_t *handle*, **int32_t *_error_);**

void EVENT_unregister_fd(int32_t *epfd*, **int32_t** *fd*, **int32_t *_error_);**

void EVENT_wait(int32_t *epfd*, **int32_t *_fd_, **int32_t *_event_,
int32_t *_handle_, **int32_t *timeoutms*, **int32_t *_error_);**

Link with **-lsimpleio**.

DESCRIPTION

All functions return either **0** (upon success) or an **errno** value (upon failure) in **error*.

EVENT_open() must be called before any of the other functions, to open a connection to the **epoll** event notification subsystem.

EVENT_close() must be called to close the connection to the **epoll** subsystem.

EVENT_register_fd() registers the file descriptor *fd* for the event notifications selected by the *events* parameter. Event notification codes, such as **EPOLLIN** (input ready), are defined in **/usr/include/sys/epoll.h**, and may be **OR**'d together to register for more than one type of event notification. The *handle* parameter is passed in to the Linux kernel and will be passed back to **EVENT_wait()** when an event notification occurs.

EVENT_modify_fd() modifies the event notifications enabled on a previously registered file descriptor. The most common use case is to rearm a file descriptor registered with **EPOLLONESHOT** for further event notifications. After such a file descriptor has delivered an event, it will be disabled from delivering any further event notifications until it is rearmed. The *handle* parameter is passed in to the Linux kernel and will be passed back to **EVENT_wait()** when an event notification occurs.

EVENT_unregister_fd() unregisters event notifications for the file descriptor *fd*.

EVENT_wait() waits until an event notification occurs for any of the previously registered file descriptors. The *timeoutms* parameter indicates the time in milliseconds to wait for an event notification. A value of zero indicates **EVENT_wait()** should return immediately whether or not an event notification is available. If an event notification occurs before the timeout expires, **error* will be set to **0**, **fd* and **event* will be set to the next available file descriptor and event notification code, and **handle* will be set to whatever value was supplied to **EVENT_register_fd()** or **EVENT_modify_fd()**. If no event notification occurs before the timeout expires, **error* will be set to **EAGAIN** and **fd*, **event*, and **handle* will all be set to zero. If some other error occurs, **error will be set to an* **errno** value and **fd*, **event*, and **handle* will all be set to zero.

SEE ALSO

libsimpleio(2), **libadc**(2), **libdac**(2), **libgpio**(2), **libhidraw**(2),
libi2c(2), **libipv4**(2), **liblinux**(2), **liblinx**(2), **libpwm**(2),
libserial(2), **libspi**(2), **libstream**(2), **libwatchdog**(2)

AUTHOR

Philip Munts, President, Munts AM Corp dba Munts Technologies

NAME

libgpio −− Linux Simple I/O Library: GPIO Module

SYNOPSIS

#include <libsimpleio/libgpio.h>

void GPIO_chip_info(int32_t *chip*, **char ***name*, **int32_t** *namesize*,
char ***label*, **int32_t *labelsize*, **int32_t ***lines*, **int32_t ***error*);

void GPIO_line_info(int32_t *chip*, **int32_t** *line*, **int32_t ***flags*,
char ***name*, **int32_t *namesize*, **char ***label*, **int32_t** *labelsize*,
int32_t *error*);

void GPIO_line_open(int32_t *chip*, **int32_t** *line*, **int32_t** *flags*,
int32_t *events*, **int32_t** *state*, **int32_t *** *fd*, **int32_t ***error*);

void GPIO_line_close(int32_t *fd*, **int32_t ***error*);

void GPIO_line_read(int32_t *fd*, **int32_t ***state*, **int32_t ***error*);

void GPIO_line_write(int32_t *fd*, **int32_t** *state*, **int32_t ***error*);

void GPIO_line_event(int32_t *fd*, **int32_t ***state*, **int32_t ***error*);

Deprecated sysfs API:

```
typedef enum
{
 GPIO_DIRECTION_INPUT,
 GPIO_DIRECTION_OUTPUT,
} GPIO_DIRECTION_t;

typedef enum
{
 GPIO_EDGE_NONE,
 GPIO_EDGE_RISING,
 GPIO_EDGE_FALLING,
 GPIO_EDGE_BOTH
} GPIO_EDGE_t;

typedef enum
{
 GPIO_POLARITY_ACTIVELOW,
 GPIO_POLARITY_ACTIVEHIGH,
} GPIO_POLARITY_t;
```

void GPIO_configure(int32_t *pin*, **int32_t** *direction*, **int32_t** *state*,
int32_t *edge*, **int32_t** *polarity*, **int32_t ***error*);

void GPIO_open(int32_t *pin*, **int32_t *** *fd*, **int32_t ***error*);

void GPIO_close(int32_t *fd*, **int32_t ***error*);

void GPIO_read(int32_t *fd*, **int32_t ***state*, **int32_t ***error*);

void GPIO_write(int32_t *fd***, int32_t** *state***, int32_t ****error***);**

Link with **-lsimpleio**.

DESCRIPTION

All functions return either **0** (upon success) or an **errno** value (upon failure) in **error*.

GPIO_chip_info() retrieves information about a particular GPIO chip (subsystem). The *chip* parameter selects the particular GPIO chip. Information strings are returned in *name* and *label* (with maximum sizes indicated by *namesize* and *labelsize*) and the number of GPIO lines (pins) available from this chip is returned in *lines*.

GPIO_line_info() retrieves information about a particular GPIO pin. The *chip* and *line* parmeters select the particular GPIO pin. Current configuration flags for the GPIO pin are returned in *flags*. Information strings are returned in *name* and *label* (with maximum sizes indicated by *namesize* and *labelsize*).

GPIO_line_open() configures and opens a single GPIO pin. The *chip* and *line* parmeters select the particular GPIO pin. Requested configuration flags are passed in *flags*, requested event (interrupt edge) flags are passed in *events*, and the initial state (**0** or **1**) for output pins is passed in *state*. Upon success, a file descriptor for the GPIO pin will be returned in **fd*.

GPIO_line_close() closes a previously opened GPIO pin.

GPIO_line_read() gets the current state of a GPIO pin. The state (**0** or **1**) of the GPIO pin will be returned in **state*.

GPIO_line_write() sets a GPIO pin output state. The new state (**0** or **1**) is passed in the *state* parameter.

GPIO_line_event() waits (blocking the calling process) for an interrupt on an GPIO interrupt input pin (configured with **GPIOHANDLE_REQUEST_INPUT** in the *flags* parameter and **GPIOEVENT_REQUEST_RISING_EDGE**, **GPIOEVENT_REQUEST_FALLING_EDGE**, or **GPIOEVENT_REQUEST_BOTH_EDGES** in the *events* parameter to *GPIO_line_open()*). The state (**0** or **1**) of the GPIO pin **after** the interrupt will be returned in **state*.

Deprecated sysfs API:

GPIO_configure() configures a single GPIO pin. The *pin* parameter selects the GPIO pin (as numbered by the Linux kernel) to be configured. The *direction* parameter may be **GPIO_DIRECTION_INPUT** or **GPIO_DIRECTION_OUTPUT**. For input pins, the *state* parameter must be **0**. For output pins, the *state* parameter may be **0** or **1** to set the initial state. For input pins, the *edge* parameter may be **GPIO_EDGE_NONE**, **GPIO_EDGE_RISING**, **GPIO_EDGE_FALLING**, or **GPIO_EDGE_BOTH**. For output pins, the *edge* parameter must be **GPIO_EDGE_NONE.** The *polarity* parameter may be **GPIO_POLARITY_ACTIVELOW** or **GPIO_POLARITY_ACTIVEHIGH**.

The **udev** rules included in the **libsimpleio** package will create a symbolic link from **/dev/gpioxx** to **/sys/class/gpio/gpioxx/value** when a GPIO pin is configured.

GPIO_open() opens a GPIO pin device. The GPIO pin number is passed in the *pin* parameter. Upon success, a file descriptor for the GPIO pin device is returned in **fd*.

GPIO_close() closes a previously opened GPIO pin device.

GPIO_read() gets the current state of a GPIO pin. Upon success, the current state (**0** or **1**) of the GPIO pin will be returned in **state*.

GPIO_write() sets a GPIO pin output state. The new state (**0** or **1**) is passed in the *state* parameter.

SEE ALSO

libsimpleio(2), **libadc**(2), **libdac**(2), **libevent**(2), **libhidraw**(2),
libi2c(2), **libipv4**(2), **liblinux**(2), **liblinx**(2), **libpwm**(2),
libserial(2), **libspi**(2), **libstream**(2), **libwatchdog**(2)

AUTHOR

Philip Munts, President, Munts AM Corp dba Munts Technologies

NAME

libhidraw −− Linux Simple I/O Library: Raw HID Module

SYNOPSIS

#include <libsimpleio/libhidraw.h>

void HIDRAW_open1(const char ****name*, int32_t ****fd***, int32_t ****error***);**

void HIDRAW_open2(int32_t *VID*, **int32_t** *PID*, **int32_t ****fd*,
int32_t ****error*);**

void HIDRAW_open3(int32_t *VID*, **int32_t** *PID*, **const char ****serial*,
int32_t ****fd*, **int32_t ****error*);**

void HIDRAW_close(int32_t *fd*, **int32_t ****error***);**

void HIDRAW_get_name(int32_t *fd*, **char ****name*, **int32_t** *namesize*,
int32_t ****error*);**

void HIDRAW_get_info(int32_t *fd*, **int32_t ****bustype*, **int32_t ****vendor*,
int32_t ****product*, **int32_t ****error*);**

void HIDRAW_send(int32_t *fd*, **void ****buf*, **int32_t** *bufsize*,
int32_t ****count*, **int32_t ****error*);**

void HIDRAW_receive(int32_t *fd*, **void ****buf*, **int32_t** *bufsize*,
int32_t ****count*, **int32_t ****error*);**

Link with **-lsimpleio**.

DESCRIPTION

All functions return either **0** (upon success) or an **errno** value (upon failure) in **error*.

HIDRAW_open1() opens a raw HID device by name. The device node name (**/dev/hidrawN**) must be passed in *name*. Upon success, a file descriptor for the raw HID device is returned in **fd*.

HIDRAW_open2() opens the first raw HID device that matches the vendor ID and product ID passed in *VID* and *PID*. Upon success, a file descriptor for the raw hid device is returned in **fd*.

HIDRAW_open3() opens the raw HID device that matches the vendor ID, product ID, and serial number passed in *VID*, *PID*, and *serial*. Upon success, a file descriptor for the raw hid device is returned in **fd*.

HIDRAW_close() closes a previously opened raw HID device. The HID device file descriptor must be passed in *fd*.

HIDRAW_get_name() fetches an information string from the raw HID device. The HID device file descriptor must be passed in *fd*. The destination buffer address must be passed in **name* and the destination buffer size must be passed in *namesize*. The minimum size for the destination buffer is 16 bytes.

HIDRAW_get_info() fetches the bus type, vendor ID and product ID from the raw HID device. The HID device file descriptor must be passed in *fd*. The bus type is returned in **bustype*, and may be **BUS_USB**, **BUS_HIL**, **BUS_BLUETOOTH** or **BUS_VIRTUAL**. These values are defined in **/usr/include/linux/input.h**. The vendor and product ID's are returned in **vendor* and **product* respectively.

HIDRAW_send() sends a message (also known as a HID report) to the raw HID device. The HID device file descriptor must be passed in *fd*. The address of the message buffer must be passed in *buf* and the size of the message buffer must be passed in *bufsize*. The message size will typically be either 64 or 65 bytes, depending on whether the particular raw HID device uses the first byte for the report number. Upon

success, the number of bytes actually sent will be returned in **count*.

HIDRAW_receive() receives a message (i.e. a HID report) from the raw HID device. The HID device file descriptor must be passed in *fd*. The address of the message buffer must be passed in *buf* and the size of the message buffer must be passed in *bufsize*. The message size will typically be either 64 or 65 bytes, depending on whether the particular raw HID device uses the first byte for the report number. Upon success, the number of bytes actually received will be returned in **count*.

SEE ALSO

libsimpleio(2), **libadc**(2), **libdac**(2), **libevent**(2), **libgpio**(2), **libi2c**(2), **libipv4**(2), **liblinux**(2), **liblinx**(2), **libpwm**(2), **libserial**(2), **libspi**(2), **libstream**(2), **libwatchdog**(2)

AUTHOR

Philip Munts, President, Munts AM Corp dba Munts Technologies

NAME

libi2c −− Linux Simple I/O Library: I^2C Module

SYNOPSIS

#include <libsimpleio/libi2c.h>

void I2C_open(const char ****name*, int32_t ****fd***, int32_t ****error***);**

void I2C_close(int32_t *fd***, int32_t ****error***);**

void I2C_transaction(int32_t *fd***, int32_t** *slaveaddr***, void ****cmd***, int32_t** *cmdlen***,**
void ****resp*, int32_t** *resplen***, int32_t ****error***);**

Link with **-lsimpleio**.

DESCRIPTION

All functions return either **0** (upon success) or an **errno** value (upon failure) in **error*.

I2C_open() opens an I^2C bus controller device. The device name, **/dev/i2c-x**, must be passed in the *name* parameter. Upon success, a file descriptor for the I^2C bus controller device is returned in **fd*.

I2C_close() closes a previously opened I^2C bus controller device.

I2C_transaction() performs a single I^2C bus transaction, with optional transmit and receive phases. The I^2C slave device address must be passed in the *slaveaddr* parameter. Either the address of a command message and its length must be passed in the *cmd* and *cmdlen* parameters, or **NULL** and **0** for a receive only transaction. Either the address of a receive buffer and its size must be passed in the *resp* and *resplen* parameters, or **NULL** and **0** for a transmit only transaction.

SEE ALSO

libsimpleio(2), **libadc**(2), **libdac**(2), **libevent**(2), **libgpio**(2),
libhidraw(2), **libipv4**(2), **liblinux**(2), **liblinx**(2), **libpwm**(2),
libserial(2), **libspi**(2), **libstream**(2), **libwatchdog**(2)

AUTHOR

Philip Munts, President, Munts AM Corp dba Munts Technologies

NAME

libipv4 −− Linux Simple I/O Library: IPv4 TCP Module

SYNOPSIS

#include <libsimpleio/libipv4.h>

void IPV4_resolve(char *_name_, int32_t *_addr_, int32_t *_error_);

void IPV4_ntoa(int32_t _addr_**, char *_dst_, int32_t** _dstsize_**, int32_t *_error_);**

void TCP4_connect(int32_t _addr_**, int32_t** _port_**, int32_t *** _fd_**, int32_t *_error_);**

void TCP4_accept(int32_t _addr_**, int32_t** _port_**, int32_t *** _fd_**, int32_t *_error_);**

void TCP4_server(int32_t _addr_**, int32_t** _port_**, int32_t *** _fd_**, int32_t *_error_);**

void TCP4_close(int32_t _fd_**, int32_t *_error_);**

void TCP4_send(int32_t _fd_**, void *_buf_, int32_t** _bufsize_**, int32_t *_count_,**
int32_t *_error_);

void TCP4_receive(int32_t _fd_**, void *_buf_, int32_t** _bufsize_**, int32_t *_count_,**
int32_t *_error_);

void UDP4_open(int32_t _addr_**, int32_t** _port_**, int32_t *** _fd_**, int32_t *_error_);**

void UDP4_close(int32_t _fd_**, int32_t *_error_);**

void UDP4_send(int32_t _fd_**, int32_t** _addr_**, int32_t** _port_**, void *_buf_,**
int32_t _bufsize_**, int32_t** _flags_**, int32_t *_count_, int32_t *_error_);**

void UDP4_receive(int32_t _fd_**, int32_t *_addr_, int32_t *_port_, void *_buf_,**
int32_t _bufsize_**, int32_t** _flags_**, int32_t *_count_, int32_t *_error_);**
Link with **-lsimpleio**.

DESCRIPTION

All functions return either **0** (upon success) or an **errno** value (upon failure) in *_error_.

IPV4_resolve() attempts to resolve an IPv4 address string passed in *_name_ (containing a domain name, a local host name like **localhost** or a dotted decimal address like **1.2.3.4**). Upon success, the 32-bit IPv4 address will be returned in *_addr_.

IPV4_ntoa() converts an IPv4 address to a dotted decimal address string. The address of the destination buffer is passed in *_dst_ and its size, which must be at least 16 bytes, is passed in _dstsize_. Upon success, the dotted decimal address string will be returned in *_dst_.

TCP4_connect() attempts to connect to a IPv4 TCP server. The 32-bit IPv4 address is passed in _addr_ and the 16-bit TCP port number is passed in _port_. Upon successful connection, a stream file descriptor will be returned in *_fd_.

TCP4_accept() waits for an incoming connection request from a IPv4 TCP client. Either **INADDR_ANY** may be passed in _addr_ to bind to (i.e. listen on) all network interfaces, or the 32-bit IPV4 address of a particular network interface may be passed to bind to only that interface. The 16-bit TCP port number is passed in _port_. Upon successful connection, a stream file descriptor will be returned in *_fd_.

TCP4_server() operates like **TCP4_accept()** except that upon successful connection, the original server process forks to create a new and separate connection handler process. The server process continues to listen for more connection requests, without returning from **TCP4_server(),** while in the new connection handler process, **TCP4_server()** does return, with a stream file descriptor for the new connection returned

in **fd*.

TCP4_close() closes a previously opened IPv4 TCP stream. The stream file descriptor is passed in *fd*.

TCP4_send() sends data to a IPv4 TCP stream. The stream file descriptor is passed in *fd*. The transmit buffer address is passed in *buf* and its size is passed in *bufsize*. Upon success, the number of bytes actually sent will be returned in **count*.

TCP4_receive() receives data from a IPv4 TCP stream. The stream file descriptor is passed in *fd*. The receive buffer address is passed in *buf* and its size is passed in *bufsize*. Upon success, the number of bytes actually received will be returned in **count*.

UDP4_open() opens a UDP datagram socket. The IP address of the network interface to bind the socket to is passed in the *addr* parameter. A value of zero or **INADDR_ANY** indicates the socket should bind to all network interfaces. The UDP port number is passed in the *port* parameter. A value of zero indicates the kernel should automatically select a port number. Upon successful completion, a UDP socket file descriptor will be returned in **fd*.

UDP4_close() closes a previously opened UDP socket. The UDP socket file descriptor is passed in *fd*.

UDP4_send() sends a UDP datagram. The UDP socket file descriptor is passed in *fd*. The IPv4 address of the destination network node is passed in *addr*. The UDP port number of the destination network node is passed in *port*. The UDP datagram is passed in **buf* and its size in bytes is passed in *bufsize*. Flags for the **sendto()** Linux system call are passed in *flags*. The number of bytes sent will be returned in **count*.

UDP4_receive() receives a UDP datagram. The UDP socket file descriptor is passed in *fd*. The IPv4 address of the source network node will be returned in **addr*. The UDP port number of the source network node will be returned in **port*. The UDP datagram will be returned in **buf*. The size of the datagram buffer will be passed in *bufsize*. Flags for the **recvfrom()** Linux system call are passed in *flags*. The number of bytes received will be returned in **count*.

SEE ALSO

libsimpleio(2), **libadc**(2), **libdac**(2), **libevent**(2), **libgpio**(2), **libhidraw**(2), **libi2c**(2), **liblinux**(2), **liblinx**(2), **libpwm**(2), **libserial**(2), **libspi**(2), **libstream**(2), **libwatchdog**(2)

AUTHOR

Philip Munts, President, Munts AM Corp dba Munts Technologies

NAME

liblinux −− Linux Simple I/O Library: Linux System Call Wrapper Module

SYNOPSIS

#include <libsimpleio/liblinux.h>

void LINUX_detach(int32_t *_error_);

void LINUX_drop_privileges(const char *_username_, int32_t *_error_);

void LINUX_openlog(const char *_id_, int32_t _options_, int32_t _facility_,
int32_t *_error_);

void LINUX_syslog(int32_t _priority_, const char *_msg_, int32_t *_error_);

int32_t LINUX_errno(void);

void LINUX_strerror(int32_t _error_, char *_buf_, int32_t _bufsize_);

void LINUX_poll(int32_t _numfiles_, int32_t *_files_, int32_t *_events_,
int32_t *_results_, int32_t _timeout_, int32_t *_error_);

void LINUX_usleep(int32_t _microseconds_, int32_t *_error_);

void LINUX_command(const char *_cmd_, int32_t *_status_, int32_t *_error_);

void LINUX_popen_read(const char *_cmd_, void **_stream_, int32_t *_error_);

void LINUX_popen_write(const char *_cmd_, void **_stream_, int32_t *_error_);

void LINUX_pclose(void *_stream_, int32_t *_error_);

Link with **-lsimpleio**.

DESCRIPTION

These functions wrap certain useful Linux system calls for use by other programming languages such as Ada, C#, Java, and Pascal. They are provided for the convenience of devlopers using **libsimpleio** with those langauges. For the C programming language, they offer no particular benefit over the regular system call wrappers.

All functions return either **0** (upon success) or an **errno** value (upon failure) in **error*.

LINUX_detach() detaches the calling process from its controlling terminal and continues execution in the background.

LINUX_drop_privileges() allows a process started by the superuser to drop its privileges to those of the user specified by the *username* parameter.

LINUX_openlog() opens a connection to the **syslog** message logger. The *options* and *facility* parameters accept the same values as the **openlog()** system library function.

LINUX_syslog() transmits a text message supplied in the *msg* parameter to the **syslog** message logger. The *priority* parameter accepts the same values as the **syslog()** system library function.

LINUX_errno() returns the current value of the **errno** variable.

LINUX_strerror() retrieves the error message for the **errno** value passed in the *error* parameter. A destination buffer address and size must be passed in the *buf* and *bufsize* parameters.

LINUX_poll() waits for events on an set of file descriptors. The number of file descriptors is passed in the

numfiles parameter. An array of file descriptors is passed in the *files* parameter. An array of requested events, using the same values as the **poll()** system library function, is passed in the *events* parameter. The results for each file descriptor, also using the same values as the **poll()** system library function, will be returned in the *results* parameter. A timeout in milliseconds may be passed in the *timeout* parameter. A timeout of zero indicates an immediate return without waiting at all. A timeout of **-1** indicates waiting forever. If the timeout expires without any event on any file descriptor occurring, then *error* will be set to **EAGAIN**.

LINUX_usleep() causes the calling program to sleep for the period indicated by *microseconds*, by calling the **usleep()** system library function.

LINUX_command() passes the shell command string specified by the *cmd* parameter to the **system()** system library function for execution. The return value from the shell command will be returned in *status*.

LINUX_popen_read() opens a pipe for reading standard output from another program. The shell command string for running the program must be passed in the *cmd* parameter. The **FILE** pointer for the pipe will be returned in the *stream* parameter.

LINUX_popen_read() opens a pipe for writing standard input to another program. The shell command string for running the program must be passed in the *cmd* parameter. The **FILE** pointer for the pipe will be returned in the *stream* parameter.

LINUX_pclose() closes a pipe opened by **LINUX_popen_read()** or **Linux_popen_close()**. The **FILE** pointer for the pipe must be passed in the *stream* parameter.

SEE ALSO

libsimpleio(2), **libadc**(2), **libdac**(2), **libevent**(2), **libgpio**(2), **libhidraw**(2), **libi2c**(2), **libipv4**(2), **liblinx**(2), **libpwm**(2), **libserial**(2), **libspi**(2), **libstream**(2), **libwatchdog**(2)

AUTHOR

Philip Munts, President, Munts AM Corp dba Munts Technologies

NAME

liblinx −− Linux Simple I/O Library: LabView LINX Remote I/O Module

SYNOPSIS

#include <libsimpleio/liblinx.h>

Structures:

```
typedef struct
{
 uint8_t  SoF;
 uint8_t  PacketSize;
 uint16_t PacketNum;
 uint16_t Command;
 uint8_t  Args[54];
} LINX_command_t;

typedef struct
{
 uint8_t  SoF;
 uint8_t  PacketSize;
 uint16_t PacketNum;
 uint8_t  Status;
 uint8_t  Data[55];
} LINX_response_t;
```

Server Routines:

void LINX_receive_command(int32_t *fd*, **LINX_command_t ***cmd*, **int32_t ***count*, **int32_t ***error*);

void LINX_transmit_response(int32_t *fd*, **LINX_response_t ***resp*, **int32_t ***error*);

Client Routines:

void LINX_transmit_command(int32_t *fd*, **LINX_command_t ***cmd*, **int32_t ***error*);

void LINX_receive_response(int32_t *fd*, **LINX_response_t ***resp*, **int32_t ***count*, **int32_t ***error*);

Byte Packing Routines:

uint16_t LINX_makeu16(uint8_t *b0*, **uint8_t** *b1*);

uint32_t LINX_makeu32(uint8_t *b0*, **uint8_t** *b1,* **uint8_t** *b2*, **uint8_t** *b3*);

Byte Unpacking Routines:

uint8_t LINX_splitu16(uint16_t *u16*, **int32_t** *bn*);

uint8_t LINX_splitu32(uint32_t *u32*, **int32_t** *bn*);

Link with **-lsimpleio**.

DESCRIPTION

These routines perform framing and encoding or decoding of LabView LINX remote I/O commands (from client to server) and responses (from server to client) to and from a bidirectional byte stream, which will typically be either a serial port or a network socket.

The transmit routines encode a frame from a command or response structure and write it to the stream indicated by the stream file descriptor *fd*. If the frame was written to the stream successfully, **error* will be set to zero, otherwise it will be set to an **errno** value.

The receive routines read exactly one byte from the stream indicated by the stream file descriptor *fd*. If the byte was read successfully and completes a frame, **error* will be set to zero. If a byte was read successfully, but did not complete a frame, **error* will be set to **EAGAIN.** If the read failed, **error* will be set to an **errno** value and the previous data discarded. Successive calls to each receive routine must pass the same command or response structure. The **count* parameter preserves a byte counter between successive function calls.

A LINX server running on some hardware device will typically have a message loop that calls **LINX_receive_command()** to get each command from the LINX client, do some work, and then call **LINX_transmit_response()** to return results to the client.

A LINX client will typically call **LINX_transmit_command()** to send each command to the server and immediately thereafter call **LINX_receive_response()** to receive the results from the server.

The byte packing routines **LINX_makeu16() and LINX_makeu32()** pack two or four unsigned bytes into a 16-bit or 32-bit unsigned integer. *b0* is the most significant byte and *b1* or *b3* is the least significant byte.

The byte unpacking routines **LINX_splitu16() and LINX_splitu32()** return a signle unsigned byte of a 16-bit or 32-bit unsigned integer, selected by the *bn* byte index parameter. A byte index of **0** selects the most significant byte and a byte index of **1** or **3** selects the least significant byte.

SEE ALSO

libsimpleio(2), **libadc**(2), **libdac**(2), **libevent**(2), **libgpio**(2), **libhidraw**(2), **libi2c**(2), **libipv4**(2), **liblinux**(2), **libpwm**(2), **libserial**(2), **libspi**(2), **libstream**(2), **libwatchdog**(2)

https://www.labviewmakerhub.com/doku.php?id=learn:libraries:linx:spec:start

AUTHOR

Philip Munts, President, Munts AM Corp dba Munts Technologies

NAME

libpwm −− Linux Simple I/O Library: PWM Output Module

SYNOPSIS

#include <libsimpleio/libpwm.h>

typedef enum
{
PWM_POLARITY_ACTIVELOW,
PWM_POLARITY_ACTIVEHIGH,
} PWM_POLARITY_t;

void PWM_configure(int32_t *chip*, **int32_t** *channel*, **int32_t** *period*,
int32_t *ontime*, **int32_t** *polarity*, **int32_t** ***error*);

void PWM_open(int32_t *chip*, **int32_t** *channel*, **int32_t** **fd*,
int32_t ***error*);

void PWM_close(int32_t *fd*, **int32_t** ***error*);

void PWM_write(int32_t *fd*, **int32_t** *ontime*, **int32_t** ***error*);

Link with **-lsimpleio**.

DESCRIPTION

All functions return either **0** (upon success) or an **errno** value (upon failure) in **error*.

PWM_configure() configures a single PWM output. The *chip* parameter selects the PWM controller chip (as numbered by the Linux kernel) and the *channel* parameter selects the PWM output (also as numbered by the Linux kernel) to be configured. The *period* parameter sets the PWM output pulse period in nanoseconds. Note that many PWM controllers require the same PWM pulse frequency for all channels. Therefore, configuring different pulse period values for different channels within the same PWM controller may result in incorrect operation. The *ontime* parameter sets the initial PWM output pulse width in nanoseconds. The *polarity* parameter sets the PWM output polarity and may be **PWM_POLARITY_ACTIVELOW** or **PWM_POLARITY_ACTIVEHIGH**. Note that some PWM controllers will not allow the **PWM_POLARITY_ACTIVELOW** setting.

PWM_open() opens a (previously) configured PWM output device. The PWM controller chip number must be passed in the *chip* parameter and the PWM output number must be passed in the *channel* parameter. Upon success, a file descriptor for the PWM output device is returned in **fd*.

PWM_close() closes a previously opened PWM output device.

PWM_write() changes the PWM output pulse width. The file descriptor for an open PWM output device must be passed in the *fd* parameter. The new PWM output pulse width in nanoseconds must be passed in the *ontime* parameter.

SEE ALSO

libsimpleio(2), **libadc**(2), **libdac**(2), **libevent**(2), **libgpio**(2), **libhidraw**(2), **libi2c**(2), **libipv4**(2), **liblinux**(2), **liblinx**(2), **libserial**(2), **libspi**(2), **libstream**(2), **libwatchdog**(2)

AUTHOR

Philip Munts, President, Munts AM Corp dba Munts Technologies

NAME

libserial −− Linux Simple I/O Library: Asynchronous Serial Port Module

SYNOPSIS

#include <libsimpleio/libserial.h>

typedef enum
{
SERIAL_PARITY_NONE,
SERIAL_PARITY_EVEN,
SERIAL_PARITY_ODD,
} SERIAL_PARITY_t;

void SERIAL_open(const char * *name*, **int32_t** *baudrate*, **int32_t** *parity*,
int32_t *databits*, **int32_t** *stopbits*, **int32_t *** *fd*, **int32_t *** *error*);

void SERIAL_close(int32_t *fd*, **int32_t *** *error*);

void SERIAL_send(int32_t *fd*, **void *** *buf*, **int32_t** *bufsize*,
int32_t * *count*, **int32_t *** *error*);

void SERIAL_receive(int32_t *fd*, **void *** *buf*, **int32_t** *bufsize*,
int32_t * *count*, **int32_t *** *error*);

Link with **-lsimpleio**.

DESCRIPTION

All functions return either **0** (upon success) or an **errno** value (upon failure) in **error*.

SERIAL_open() opens and configures a serial port device. The device name must be passed in the *name* parameter. The *baudrate* parameter sets the serial port bit rate. Allowed values are **50**, **75**, **110**, **134**, **150**, **200**, **300**, **600**, **1200**, **1800**, **2400**, **4800**, **9600**, **19200**, **38400**, **57600**, **115200**, **230400**, **460800**, **500000**, **576000**, **921600**, and **1000000**. The *parity* parameter sets the parity mode. Allowed values are **SERIAL_PARITY_NONE**, **SERIAL_PARITY_EVEN**, and **SERIAL_PARITY_ODD**. The *databits* parameter sets the number of bits per character, and may be from **5** to **8**. The *stopbits* parameter sets the number of stop bits per character, and may be **1** or **2**. Upon success, a file descriptor for the serial port device is returned in **fd*.

SERIAL_close() closes a previously opened serial port device. The serial port device file descriptor is passed in *fd*.

SERIAL_send() sends data to the serial port device. The serial port device file descriptor is passed in *fd*. The transmit buffer address is passed in *buf* and its size is passed in *bufsize*. Upon success, the number of bytes actually sent will be returned in **count*.

SERIAL_receive() receives data from the serial port device. The serial port device file descriptor is passed in *fd*. The receive buffer address is passed in *buf* and its size is passed in *bufsize*. Upon success, the number of bytes actually received will be returned in **count*.

SEE ALSO

libsimpleio(2), **libadc**(2), **libdac**(2), **libevent**(2), **libgpio**(2), **libhidraw**(2), **libi2c**(2), **libipv4**(2), **liblinux**(2), **liblinx**(2), **libpwm**(2), **libspi**(2), **libstream**(2), **libwatchdog**(2)

AUTHOR

Philip Munts, President, Munts AM Corp dba Munts Technologies

NAME

libspi −− Linux Simple I/O Library: SPI Module

SYNOPSIS

#include <libsimpleio/libspi.h>

void SPI_open(const char ****name*, int32_t** *mode***, int32_t** *wordsize***,**
int32_t *speed***, int32_t ****fd***, int32_t ****error***);**

void SPI_close(int32_t *fd***, int32_t ****error***);**

void SPI_transaction(int32_t *spifd***, int32_t** *csfd***, void ****cmd***,**
int32_t *cmdlen***, int32_t** *delayus***, void ****resp***, int32_t** *resplen***,**
int32_t ****error*);**

Link with **-lsimpleio**.

DESCRIPTION

All functions return either **0** (upon success) or an **errno** value (upon failure) in **error*.

SPI_open() opens an SPI slave device. The device name, **/dev/spidevx.x**, must be passed in the *name* parameter. The *mode* parameter specifies the SPI bus transfer mode, **0** to **3**. The *wordsize* parameter specifies the SPI bus transfer unit size, usually **8**, **16**, or **32** bits. Some SPI controllers only allow 8-bit transfers. The *speed* parameter specifies the SPI bus transfer speed in bits per second. Upon success, a file descriptor for the SPI slave device is returned in **fd*.

SPI_close() closes a previously opened SPI slave device.

SPI_transaction() performs a single SPI bus transaction, with optional transmit and receive phases. Either the address of a command message and its length must be passed in the *cmd* and *cmdlen* parameters, or **NULL** and **0** for a receive only transaction. The *delayus* parameter indicates the time in microseconds between the transmit and receive phases. It should be set long enough for the SPI slave device to execute the command and generate its response. Either the address of a receive buffer and its size must be passed in the *resp* and *resplen* parameters, or **NULL** and **0** for a transmit only transaction. The *csfd* parameter should be set to **SPI_CS_AUTO** to use the hardware controlled slave chip select signal or set to the open file descriptor for a GPIO pin to use for the software controlled slave chip select signal.

SEE ALSO

libsimpleio(2), **libadc**(2), **libdac**(2), **libevent**(2), **libgpio**(2),
libhidraw(2), **libi2c**(2), **libipv4**(2), **liblinux**(2), **liblinx**(2),
libpwm(2), **libserial**(2), **libstream**(2), **libwatchdog**(2)

AUTHOR

Philip Munts, President, Munts AM Corp dba Munts Technologies

NAME

libstream −− Linux Simple I/O Library: Stream Framing Protocol Module

SYNOPSIS

#include <libsimpleio/libstream.h>

typedef ssize_t (**STREAM_readfn_t***)(int** *fd***, void ****buf***, size_t** *count***);**

typedef ssize_t (**STREAM_writefn_t***)(int** *fd***, const void ****buf***,**
size_t *count***);**

void STREAM_change_readfn(STREAM_readfn_t *newread***, int32_t ****error***);**

void STREAM_change_writefn(STREAM_writefn_t *newwrite***, int32_t ****error***);**

void STREAM_encode_frame(void ****src*, int32_t** *srclen***, void ****dst***,**
int32_t *dstsize***, int32_t ****dstlen***, int32_t ****error***);**

void STREAM_decode_frame(void ****src*, int32_t** *srclen***, void ****dst***,**
int32_t *dstsize***, int32_t ****dstlen***, int32_t ****error***);**

void STREAM_send_frame(int32_t *fd***, void ****buf***, int32_t** *bufsize***,**
int32_t ****count*, int32_t ****error***);**

void STREAM_receive_frame(int32_t *fd***, void ****buf***, int32_t** *bufsize***,**
int32_t * *framesize***, int32_t ****error***);**

Link with **-lsimpleio**.

DESCRIPTION

These functions encode, decode, send, and receive frames to and from a bidirectional byte stream, which will typically be either a serial port or a network socket. The frames are encoded according the the Stream Framing Procotol. See below for a link to the protocol specification.

All functions return either **0** (upon success) or an **errno** value (upon failure) in **error*.

STREAM_change_readfn() changes the function used to read from the underlying stream from the default **read()** to some other compatible function.

STREAM_change_writefn() changes the function used to write to the underlying stream from the default **write()** to some other compatible function.

STREAM_encode_frame() encodes the message passed in **src*, with its length passed in *srclen*. Empty messages (*srclen*==0) are allowed. The encoded frame will be returned in **dst*, whose maximum size must be passed in *dstsize*. The size of the destination buffer must be at least 2 * *srclen* + 8 bytes. The actual size of the encoded frame will be returned in **dstlen*.

STREAM_decode_frame() decodes the frame passed in **src*, with its length passed in *srclen*. The decoded message will be returned in **dst*, whose maximum size must be passed in *dstsize*. The actual size of the decoded message will be returned in **dstlen*.

STREAM_send_frame() writes an encoded frame to the bidirectional byte stream indicated by the file descriptor *fd*. The encoded frame is passed in *buf* and its size is passed in *bufsize*. Upon success, the number of bytes actually sent will be returned in **count*.

STREAM_receive_frame() reads one byte from the bidirectional byte stream indicated by the file descriptor *fd* and attempts to assemble a frame. It should be called repeatedly with the same **buf*, *bufsize*, and **framesize* parameters. The **framesize* parameter is incremented for each byte received, and zeroed if an error occurs. The **error* parameter will be set to **EAGAIN** while a frame is being assembled. Upon successful assembly of a complete frame, its size will be returned in **framesize* and zero returned in **error*.

SEE ALSO

libsimpleio(2), **libadc**(2), **libdac**(2), **libevent**(2), **libgpio**(2), **libhidraw**(2), **libi2c**(2), **libipv4**(2), **liblinux**(2), **liblinx**(2), **libpwm**(2), **libserial**(2), **libspi**(2), **libwatchdog**(2)

http://git.munts.com/libsimpleio/doc/StreamFramingProtocol.pdf

AUTHOR

Philip Munts, President, Munts AM Corp dba Munts Technologies

NAME

libwatchdog −− Linux Simple I/O Library: Watchdog Timer Module

SYNOPSIS

#include <libsimpleio/libwatchdog.h>

void WATCHDOG_open(const char ****name*, int32_t ****fd***, int32_t ****error***);**

void WATCHDOG_close(int32_t *fd***, int32_t ****error***);**

void WATCHDOG_get_timeout(int32_t *fd***, int32_t ****timeout***, int32_t ****error***);**

void WATCHDOG_set_timeout(int32_t *fd***, int32_t** *newtimeout***,**
int32_t ****timeout*, int32_t ****error***);**

void WATCHDOG_kick(int32_t *fd***, int32_t ****error***);**

Link with **-lsimpleio**.

DESCRIPTION

All functions return either **0** (upon success) or an **errno** value (upon failure) in **error*.

WATCHDOG_open() opens a watchdog timer device. The device node name, usually **/dev/watchdog**, must be passed in the *name* parameter. Upon success, a file descriptor for the watchdog timer device is returned in **fd*.

WATCHDOG_close() closes a previously opened watchdog timer device. Note that this may result in watchdog timer expiration and subsequent system reset.

WATCHDOG_get_timeout() may be used to discover the current watchdog timeout period. Upon success, the watchdog period in seconds will be returned in **timeout*.

WATCHDOG_set_timeout() may be used to change the watchdog timeout period. The requested new timeout period in seconds must be passed in *newtimeout*. Upon success, the actual new watchdog period in seconds will be returned in **timeout*. Note that the new watchdog period may be different from that requested. For example, if the watchdog timer device has a granularity of one minute, requesting a timeout of 45 seconds will result in an actual timeout of 60 seconds. Also note that the particular watchdog timer device may not allow increasing the timeout or may not allow changing it at all.

WATCHDOG_kick() may be used to reset the watchdog timer.

SEE ALSO

libsimpleio(2), **libadc**(2), **libdac**(2), **libevent**(2), **libgpio**(2),
libhidraw(2), **libi2c**(2), **libipv4**(2), **liblinux**(2), **liblinx**(2),
libpwm(2), **libserial**(2), **libspi**(2), **libstream**(2)

AUTHOR

Philip Munts, President, Munts AM Corp dba Munts Technologies