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.

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:


```
Copyright (C)2016-2019, Philip Munts, President, Munts AM Corp.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

* Redistributions of source code must retain the above copyright notice,
  this list of conditions and the following disclaimer.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE
LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
POSSIBILITY OF SUCH DAMAGE.
```

Online Resources

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

http://git.munts.com/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/debian9

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

http://git.munts.com/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/debian9/PublicKey.txt
sudo apt-key add PublicKey.txt
```

For 64-Bit Debian Linux:

```
wget http://repo.munts.com/debian9/amd64/munts-repo.deb
sudo dpkg -i munts-repo.deb
```

For 32-Bit Debian Linux:

```
wget http://repo.munts.com/debian9/i386/munts-repo.deb
sudo dpkg -i munts-repo.deb
```

For 32-Bit Raspbian (Raspberry Pi) Linux:

```
wget http://repo.munts.com/debian9/raspbian/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/libexec
/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-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

Alternatively, or for non-Debian Linux distributions, **`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
```

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_open()`** to open a raw HID device node by name. Use **`HIDRAW_open_id()`** to open the first raw HID device to match the given vendor and product identifiers. 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_strerror()`** to retrieve the error message associated with an **`errno`** error number.

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