Open Ephys++ Communication Protocol and API Specification Version 0.0

Jonathan P. Newman, Wilson Lab, MIT

April 24, 2018

Abstract

This document specifies requirements for implementing the Open Ephys++ data acqusition system. This specification entails two basic elements: (1) Communication protocols between acqusition firmware and host software and (2) an application programming interface (API) for utilizing this communication protocol. This document is incomplete and we gratefully welcome critisms and admendments.

Contents

Intentions and capabilities

- Low latency (sub millisecond)
- High bandwidth ($>$ 1000 neural data channels)
- Bidirectional
- Acquisition and control of arbitrary of hardware components using a single communication medium
  - Support generic mixes of hardware elements
  - Generic hardware configuration
  - Generic data input stream
  - Generic data output stream
- Support multiple acquisition systems on one computer
- Cross platform
- Low level: aimed at the creation of language bindings and application-specific libraries

FPGA/Host PC communication

Communication between the acquisition board firmware and API shall occur over four communication channels:

1. Signal: Read-only, short-message, asynchronous hardware events
2. Configuration: Bidirectional, register-based, synchronous configuration setting and getting
3. Input: Read-only, asynchronous, high-bandwidth firmware to host streaming
4. Output: Write-only, asynchronous, high-bandwidth host to firmware streaming

Required characteristics of these channels are described in the following paragraphs.

Signal channel (8-bit, asynchronous, read only)

The *signal* channel provides a way for the FPGA firmware to inform host of configuration results, which may be provided with a significant delay. Additionally, it allows the host to read the device map supported by the FPGA firmware. The behavior of the signal channel is equivalent to a read-only, blocking UNIX named pipe. Signal data is framed into packets using Consistent Overhead Byte Stuffing (COBS). Within this scheme, packets are delimited using 0's and always have the following format:

```
... | PACKET_FLAG data | ...
```

where `PACKET_FLAG` is 32-bit unsigned integer with a single unique bit setting, `|` represents a packet delimiter, and `...` represents other packets. This stream can be read and ignored until a desired packet is received. Reading this stream shall block if no data is available, which allows asynchronous configuration acknowledgement. Valid `PACKET_FLAG`s are:

```
enum signal {
    NULLSIG      = (1u << 0), // Null signal, ignored by host
    CONFIGWACK   = (1u << 1), // Configuration write-acknowledgement
    CONFIGWNACK  = (1u << 2), // Configuration no-write-acknowledgement
    CONFIGRACK   = (1u << 3), // Configuration read-acknowledgement
    CONFIGRNACK  = (1u << 4), // Configuration no-read-acknowledgement
    DEVICEMAPACK = (1u << 5), // Device map start acnknowledgement
    DEVICEINST   = (1u << 6), // Device map instance
};
```

Following a hardware reset, the signal channel is used to provide the device map to the host using the following packet sequence:

```
... | DEVICEMAPACK, uint32_t num_devices | DEVICEINST device dev_0
    | DEVICEINST device dev_1 | ... | DEVICEINST device dev_n | ...
```

Following a device register read or write (see configuration channel), ACK or NACK signals are pushed onto the signal stream by the firmware. For instance, on a successful register read:

```
... | CONFIGRACK, uint32_t register value | ...
```

Configuration channel (32-bit, synchronous, read and write)

The *configuration* channel supports seeking to, reading, and writing a set of configuration registers. Its behavior is equivalent to that of a normal UNIX file. There are two classes of registers handled by the configuration channel: the first set of registers encapsulates a generic device register programming interface. The remaining registers are for global context control and configuration and provide access to acquisition parameters and state control. `SEEK` locations of each configuration register, relative to the start of the stream, should be hard-coded into the API implementation file and used in the background to manipulate register state.

Device register programming interface

The device programming interface is composed of the following configuration channel registers:

- `uint32_t config_device_id`: Device ID register. Specify a device endpoint as enumerated by the firmware (e.g. an Intan chip, or a IMU chip) and to which communication will be directed using `config_reg_addr` and `config_reg_value`, as described below.
- `uint32_t config_reg_addr`: The register address of configuration to be written
- `uint32_t config_reg_value`: configuration value to be written to or read from and that corresponds to `config_reg_addr` on device `config_device_id`
- `uint32_t config_rw`: A flag indicating if a read or write should be performed. 0 indicates read operation. A value > 0 indicates write operation.
- `uint32_t config_trig`: Set > 0 to trigger either register read or write operation depending on the state of `config_rw`. If `config_rw` is 0, a read is performed. In this case `config_reg_value` is updated with value stored at `config_reg_addr` on device at `config_device_id`. If `config_rw` is 1, `config_reg_value` is written to register at `config_reg_addr` on device `config_device_id`. The `config_trig` register is always be set low by the firmware following transmission even if it is not successful or does not make sense given the address register values.

Appropriate values of `config_reg_addr` and `config_reg_value` are determined by:

- Looking at a device's data sheet if the device is an integrated circuit
- Examining the open ephys++ devices header file (oedevices.h) which contains off register addresses and descriptions for devices officially supported by this project (device id < 10000).

When a host requests a device register *read*, the following following actions take place:

1. The value of `config_trig` is checked.
    - If it is 0x00, the function call proceeds.
    - Else, the function call returns with an error specifying a retrigger.
2. `dev_idx` is copied to the `config_device_id` register on the host FPGA.
3. `addr` is copied to the `config_reg_addr` register on the host FPGA.
4. The `config_rw` register on the host FPGA is set to 0x00.
5. The `config_read_trig` register on the host FPGA is set to 0x01, triggering configuration transmission by the firmware.
6. (Firmware) A configuration read is performed by the firmware.
7. (Firmware) `config_trig` is set to 0x00 by the firmware.
8. (Firmware) `CONFIGRACK` is pushed onto the signal stream by the firmware.
9. The signal stream is pumped until either `CONFIGRACK` or `CONFIGRNACK` is received indicating that the host FPGA has either:
    - Completed reading the specified device register and copied its value to the `config_reg_value` register.
    - Failed to read the register in which case the value of `config_reg_value` contains garbage.

When a host requests a device register *write*, the following following actions take place:

1. The value of `config_trig` is checked.
    - If it is 0x00, the function call proceeds.
    - Else, the function call returns with an error specifying a retrigger.
2. `dev_idx` is copied to the `config_device_id` register on the host FPGA.
3. `addr` is copied to the `config_reg_addr` register on the host FPGA.
4. `value` is copied to the `config_reg_value` register on the host FPGA.
5. The `config_rw` register on the host FPGA is set to 0x01.
6. The `config_trig` register on the host FPGA is set to 0x01, triggering configuration transmission by the firmware.
7. (Firmware) A configuration write is performed by the firmware.
8. (Firmware) `config_trig` is set to 0x00 by the firmware.
9. (Firmware) `CONFIGWACK` is pushed onto the signal stream by the firmware.

10. The signal stream is pumped until either `CONFIGWACK` or `CONFIGwNACK` is received indicating that the host FPGA has either:
    - Successfully completed writing the specified device register
    - Failed to write the register

Following successful or unsuccessful device register read or write, the appropriate ACK or NACK packets *must* be passed to the signal channel. If they are not, the register read and write calls will block indefinitely.

Global acquisition registers

The following global acquisition registers provide information about, and control over, the entire acquisition system:

- `uint32_t running`: set to > 0 to run the system clock and produce data. Set to 0 to stop the system clock and therefore stop data flow. Results in no other configuration changes.
- `uint32_t reset`: set to > 0 to trigger a hardware reset and send a fresh device map to the host and reset hardware to its default state. Set to 0 by host firmware upon entering the reset state.
- `uint32_t sys_clock_hz`: A read-only register specifying the base hardware clock frequency in Hz. The clock counter in the read frame header is incremented at this frequency.

Data input channel (32-bit, asynchronous, read-only)

The *data input* channel provides high bandwidth communication from the FPGA firmware to the host computer using direct memory access (DMA). From the host's perspective, its behavior is equivalent to a read-only, blocking UNIX named pipe with the exception that data can only be read on 32-bit, instead of 8-bit, boundaries. The data input channel communicates with the host using frames with a read-header ("read-frames") .Read-frames are pushed into the data input channel at a rate dictated by the FPGA firmware. It is incumbent on the host to read this stream fast enough to prevent buffer overflow. At the time of this writing, a typical implementation will allocate an input buffer that occupies a 512 MB segment of kernal RAM. Increased bandwidth demands will necessitate the creation of a user-space buffer. This change shall have no effect on the API.

Data output channel (32-bit, asynchronous, write-only)

The *data output* channel provides high bandwidth communication from the host computer to the FPGA firmware using DMA via calls. From the host's perspective, its behavior is equivalent to a write-only, blocking UNIX named pipe with the exception that data can only be written on 32-bit, instead of 8-bit, boundaries. Its performance characteristics are largely identical to the data input channel.

Required API Types and Behavior

In the following sections we define required API datatypes and how they are used by the API to communicate with hardware. An implementation of this API, liboepcie, follows.

Context

A *context* shall hold all state required to manage single FPGA/Host communication system. This includes a map of devices being acquired from, data buffering elements, etc. API calls will typically take a context handle as the first argument and use it to reference required state information to enable communication and/or to mutate the context to reflect some function side effect (e.g. add device map information):

```
int api_function(context *ctx, ...);
```

Device

A *device* is defined as configurable piece of hardware with its own register address space (e.g. an integrated circuit) or something programmed within the firmware to emulate this (e.g. an electrical stimulation sub-circuit made to look like a Master-8). Host interaction with a device is facilitated using a device description, which should hold the following elements:

- `device_id`: Device ID number
- `read_size`: Device data read size per frame in bytes
- `num_reads`: Number of frames that must be read to construct a full sample (e.g., for row reads from camera)
- `write_size`: Device data write size per frame in bytes
- `num_writes`: Number of frames that must be written to construct a full output sample

An array of structures holding each of these entries forms a *device map*. A context is responsible for managing a single device map, which keep track of where to send and receive streaming data and configuration information during API calls. A detailed description of each of each value comprising a device instance is as follows:

1. `device_id`: Device identification number which is globally enumerated for the entire project
    - There is a single `enum` for the entire library which enumerates all possible devices that are controlled across `context` configurations. This enumeration will grow with the number of devices supported by the library.
    - e.g. A host board GPIO subcircuit is 0, Intan RHD2132 is 1, Intan RHD2164 is 2, etc.
    - Device IDs up to 9999 are reserved. Device ID 10000 and greater are free to use for custom hardware projects.
    - The use of device IDs less than 10000 not specified within this enumeration will result in OE_EDEVID errors.
    - Device numbers greater than 1000 are allowed for general purpose use and will not be verified by the API.
    - Incorporation into the official device enum (device IDs < 10000) can be achieved via pull-request to this repo.
2. `read_size`: Number of bytes of data transmitted by this device during a single read.
    - 0 indicates that it does not send data.
3. `num_reads`: Number of reads required to construct a full device read sample (e.g., number of columns when `read_size` corresponds to a single row of pixels from a camera sensor)
4. `write_size`: Number of bytes accepted by the device during a single write
    - 0 indicates that it does not send data.
5. `num_writes`: Number of writes required to construct a full device output sample.

Frame

A *frame* is a flat byte array containing a single sample's worth of data for a set (one to all) of devices within a device map. Data within frames is arranged into three memory sectors as follows:

```
[32 byte header,                              // 1. Header
 dev_0 idx, dev_1 idx, ... , dev_n idx,       // 2. Device map indices
 dev_0 data, dev_1 data, ... , dev_n data]    // 3. Data
```

Each frame memory sector is described below:

1. Header
    - Each frame starts with a 32-byte header
    - For reading (firmware to host) operations, the header contains
        - bytes 0-7: unsigned 64-bit integer holding system clock counter
        - bytes 8-9: unsigned 16-bit integer indicating number of devices that the frame contains data for
        - byte 10: 8-bit integer speficying fame error state. frame error. 0 = OK. 1 = data may be corrupt.
        - bytes 11-32: reserved
    - For writing (host to firmware) operations , the header contains
        - bytes 0-32: reserved
2. Device map indices
    - An array of unsigned 32-bit keys corresponding the device map captured by the host during context initialisaiton
    - The offset, size, and type information of the __i__th data block within the `data` section of each frame is determined by examining the __i__th member of the device map.
3. Data
    - Raw data blocks from each device in the device map.
    - The ordering of device-specific blocks is the same as the device index within the *device map index* portion of the frame
    - The read/write size for each device-specific block is provided in the device map
    - Perhaps in the future, data type casting information can be provided in the device map, but this is not currently required.

liboepcie: An Open Ephys ++ API Implementation

Scope and External Dependencies

liboepcie is a C library that implements the Open Ephys++ API Specification. It is written in C to facilitate cross platform and cross-language use. It is composed of two mutually exclusive file pairs:

1. oepcie.h and oepcie.c: main API implementation
2. oedevice.h and oedevices.c: officially supported device and register definitions. This file can be ignored for project that do not wish to conform to the official device specification.

liboepcie is a low level library used by high-level language binding and/or software plugin developers. It is not meant to be used by neuroscientists directly. The only external dependency aside from the C standard library is is a hardware communication backend that fulfils the requirements of the FPGA/Host Communication Specification. An example of such a backend is Xillybus, which provides proprietary FPGA IP cores and free and open source device drivers to allow the communication channels to be implemented using the PCIe bus. From the API's perspective, hardware communication abstracted to IO system calls (`open`, `read`, `write`, etc.) on file descriptors. File descriptor semantics and behavior are identical to either normal files (configuration channel) or named pipes (signal, data input, and data output channels). Because of this, a drop in replacement for the Xillybus IP Core can be used without any API changes. The development of a free and open-source FPGA cores that emulate the functionality of Xillybus would be a major benefit to the systems neuroscience community.

Importantly, the low-level synchronization, resource allocation, and logic required to use the hardware communication backend is implicit to `liboepcie` API function calls. Ochestration of the communication backend *is not directly managed by the library user.*

License

MIT

Types

Integer types

- `oe_size_t`: Fixed width size integer type.
- `oe_dev_id_t`: Fixed width device identity integer type.
- `oe_reg_addr_t`: Fixed width device register address integer type.
- `oe_reg_value_t`: Fixed width device register value integer type.

oe_ctx

Context implementation. `oe_ctx` is an opaque handle to a context structure which contains hardware and device state information.

```
// oepcie.h
typedef struct oe_ctx_impl *oe_ctx;
```

Context details are hidden in implementation file (oepcie.c):

```
typedef struct stream_fid {
    char *path;
    int fid;
} stream_fid_t;

typedef struct oe_ctx_impl {
```

```
    // Communication channels
    stream_fid_t config;
    stream_fid_t read;
    stream_fid_t write;
    stream_fid_t signal;

    // Devices
    oe_size_t num_dev;
    oe_device_t* dev_map;

    // Maximum frame sizes (bytes)
    oe_size_t max_read_frame_size;
    oe_size_t write_frame_size;

    // Data buffer
    uint8_t *buffer;
    uint8_t *buff_read_pos;
    uint8_t *buff_end_pos;

    // Acqusition state
    enum run_state {
        CTXNULL = 0,
        UNINITIALIZED,
        IDLE,
        RUNNING
    } run_state;

} oe_ctx_impl_t;
```

Each context manages a single device map. Following a hardware reset, which is triggered either by a call to `oe_init_ctx` or to `oe_set_ctx` using the `OE_RESET` option, the context `run_state` is set to UNINTIALIZED and the device map is pushed onto the signal stream by the FPGA as COBS encode packets. On the signal stream, the device map is organized as follows,

… | DEVICEMAPACK, uint32_t num_devices | DEVICEINST oe_device_t dev_0 | DEVICEINST oe_device_t dev_1 | … | DEVICEINST oe_device_t dev_n | …

where | represents '0' packet delimiters. During a call to `oe_init_ctx`, the device map is decoded from the signal stream. It can then be examined using calls to `oe_get_opt` using the `OE_DEVICEMAP` option. After the map is received, the context `run_state` becomes IDLE. A call to `oe_set_ctx` with the `OE_RUNNING` option can then be used to start acquisition by transitioning the context `run_state` to `RUNNING`.

`oe_device_t`

Device implementation. An `oe_device_t` describes one of potentially many pieces of hardware within a context. Examples include Intan chips, IMUs, optical stimulator's, camera sensors, etc. Each valid device type has a unique ID which is enumerated in the auxiliary `oedevices.h` file or some use-specific header. A map of available devices is read from hardware and stored in the current context via a call to `oe_init_ctx`. This map can be examined via calls to `oe_get_opt`.

```
typedef struct {
    oe_dev_id_t id;          // Device ID number
    oe_size_t read_size;     // Device data read size per frame in bytes
    oe_size_t num_reads;     // Number of read frames to construct a full sample
    oe_size_t write_size;    // Device data write size per frame in bytes
    oe_size_t num_writes;    // Number of written frames comprising a full sample
} oe_device_t;
```

Officially supported device IDs and configuration register definitions are provided in oedevices.h as a set of enumerations. A portion of the official device ID enumeration is defined as follows:

```c
typedef enum device_id {
    OE_IMMEDIATEIO = 0,
    OE_RHD2132,
    OE_RHD2164,
    OE_MPU9250,
    OE_ESTIM,
    ...
    OE_MAXDEVICEID = 9999
} oe_device_id_t
```

An example of a device register (for the `OE_ESTIM` device ID) enumeration is:

```c
enum oe_estim_regs {
    OE_ESTIM_NULLPARM    = 0,  // No command
    OE_ESTIM_BIPHASIC    = 1,  // Biphasic pulse (0 = monophasic, 1 = biphasic;
    OE_ESTIM_CURRENT1    = 2,  // Phase 1 current, (0 to 255 = -1.5 mA to +1.5mA)
    OE_ESTIM_CURRENT2    = 3,  // Phase 2 voltage, (0 to 255 = -1.5 mA to +1.5mA)
    OE_ESTIM_PULSEDUR1   = 4,  // Phase 1 duration, 10 microsecond steps
    OE_ESTIM_IPI         = 5,  // Inter-phase interval, 10 microsecond steps
    OE_ESTIM_PULSEDUR2   = 6,  // Phase 2 duration, 10 microsecond steps
    OE_ESTIM_PULSEPERIOD = 7,  // Inter-pulse interval, 10 microsecond steps
    OE_ESTIM_BURSTCOUNT  = 8,  // Burst duration, number of pulses in burst
    OE_ESTIM_IBI         = 9,  // Inter-burst interval, microseconds
    OE_ESTIM_TRAINCOUNT  = 10, // Pulse train duration, number of bursts in train
    OE_ESTIM_TRAINDELAY  = 11, // Pulse train delay, microseconds
    OE_ESTIM_TRIGGER     = 12, // Trigger stimulation (1 = deliver)
    OE_ESTIM_POWERON     = 13, // Control estim sub-circuit power (0 = off, 1 = on)
    OE_ESTIM_ENABLE      = 14, // Control null switch (0 = stim output shorted to ground, 1 = enabled)
    OE_ESTIM_RESTCURR    = 15, // Current between pulse phases, (0 to 255 = -1.5 mA to +1.5mA)
    OE_ESTIM_RESET       = 16, // Reset all parameters to default
};
```

These registers may be familiar to those who have used a Master-8 or pulse-pal stimulus sequencer.

`oe_frame_t`

Frame implementation. Frames are produced by calls `oe_read_frame` and provided to calls to `oe_write_frame`.

```c
typedef struct oe_frame {
    uint64_t clock;         // Base clock counter
    uint16_t num_dev;       // Number of devices in frame
    uint8_t corrupt;        // Is this frame corrupt?
    oe_size_t *dev_idxs;    // Array of device indices in frame
    oe_size_t dev_idxs_sz;  // Size in bytes of dev_idxs buffer
    oe_size_t *dev_offs;    // Device data offsets within data block
    oe_size_t dev_offs_sz;  // Size in bytes of dev_idxs buffer
    uint8_t *data;          // Multi-device raw data block
    oe_size_t data_sz;      // Size in bytes of data buffer

} oe_frame_t;
```

`oe_opt_t`

Context option enumeration. See the description of `oe_set_opt` and `oe_get_opt` for valid values.

`oe_error_t`

Error code enumeration.

```
typedef enum oe_error {
    OE_ESUCCESS         =  0,  // Success
    OE_EPATHINVALID     = -1,  // Invalid stream path, fail on open
    OE_EREINITCTX       = -2,  // Double initialization attempt
    OE_EDEVID           = -3,  // Invalid device ID on init or reg op
    OE_EREADFAILURE     = -4,  // Failure to read from a stream/register
    OE_EWRITEFAILURE    = -5,  // Failure to write to a stream/register
    OE_ENULLCTX         = -6,  // Attempt to call function w null ctx
    OE_ESEEKFAILURE     = -7,  // Failure to seek on stream
    OE_EINVALSTATE      = -8,  // Invalid operation for the current context run state
    OE_EDEVIDX          = -9,  // Invalid device index
    OE_EINVALOPT        = -10, // Invalid context option
    OE_EINVALARG        = -11, // Invalid function arguments
    OE_ECANTSETOPT      = -12, // Option cannot be set in current context state
    OE_ECOBSPACK        = -13, // Invalid COBS packet
    OE_ERETRIG          = -14, // Attempt to trigger an already triggered operation
    OE_EBUFFERSIZE      = -15, // Supplied buffer is too small
    OE_EBADDEVMAP       = -16, // Badly formated device map supplied by firmware
    OE_EBADALLOC        = -17, // Bad dynamic memory allocation
    OE_ECLOSEFAIL       = -18, // File descriptor close failure, check errno
    OE_EDATATYPE        = -19, // Invalid underlying data types
    OE_EREADONLY        = -20, // Attempted write to read only object (register, context option, etc)
    OE_ERUNSTATESYNC    = -21, // Software and hardware run state out of sync
} oe_error_t;
```

oe_create_ctx

Create a hardware context. A context is an opaque handle to a structure which contains hardware and device state information, configuration capabilities, and data format information. It can be modified via calls to `oe_set_opt`. Its state can be examined by `oe_get_opt`.

```
oe_ctx oe_create_ctx()
```

Returns `oe_ctx`

An opaque handle to the newly created context if successful. Otherwise it shall return NULL and set errno to `EAGAIN`.

Description

On success a context struct is allocated and created, and its handle is passed to the user. The context holds all state used by the library function calls for refection and hardware communication. It holds paths to FIFOs and configuration communication channels and knowledge of the hardware's parameters and run state . It is configured through calls to `oe_set_opt`. It can be examined through calls to `oe_get_opt`.

oe_init_ctx

Initialize a context, opening all file streams etc.

```
int oe_init_ctx(oe_ctx ctx)
```

Arguments

- `ctx` context

Returns `int`

- 0: success
- Less than 0: `oe_error_t`

Description

Upon a call to `oe_init_ctx`, the following actions take place

1. All required data streams are opened.
2. A device map is read from the firmware. It can be examined via calls t `oe_get_opt`.
3. The data transmission packet size is calculated and stored. It can be examined via calls t `oe_get_opt`.

Following a successful call to `oe_init_ctx`, the hardware's acquisition parameters and run state can be manipulated using calls to `oe_get_opt`.

oe_destroy_ctx

Terminate a context and free bound resources.

```
int oe_destroy_ctx(oe_ctx ctx)
```

Arguments

- `ctx` context

Returns `int`

- 0: success
- Less than 0: `oe_error_t`

Description

During context destruction, all resources allocated by `oe_create_ctx` are freed. This function can be called from any context run state. When called, an interrupt signal (TODO: Which?) is raised and any blocking operations will return immediately. Attached resources (e.g. file descriptors and allocated memory) are closed and their resources freed.

oe_get_opt

Get context options.

```
int oe_get_opt(const oe_ctx ctx, int option, void* value, size_t *size);
```

Arguments

- `ctx` context to read from
- `option` option to read
- `value` buffer to store value of `option`
- `size` pointer to the size of `value` (including terminating null character, if applicable) in bytes

Returns `int`

- 0: success
- Less than 0: `oe_error_t`

Description

The `oe_get_opt` function sets the option specified by the `option` argument to the value pointed to by the `value` argument for the context pointed to by the `ctx` argument. The `size` provides a pointer to the size of the option value in bytes. Upon successful completion `oe_get_opt` shall modify the value pointed to by `size` to indicate the actual size of the option value stored in the buffer.

Following a successful call to `oe_init_ctx`, the following socket options can be read:

`OE_CONFIGSTREAMPATH`*

Obtain path specifying config data stream.

option value type	`char *`
option description	A character string specifying the configuration stream path
default value	/dev/xillybus_oe_config_32, \\.\xillybus_oe_config_32 (Windows)

`OE_READSTREAMPATH`*

Obtain path specifying input data stream.

option value type	`char *`
option description	A character string specifying the input stream path
default value	/dev/xillybus_oe_input_32 \\.\xillybus_oe_input_32 (Windows)

`OE_WRITESTREAMPATH`*

Obtain path specifying input data stream.

option value type	`char *`
option description	A character string specifying the output stream path
default value	/dev/xillybus_oe_output_32, \\.\xillybus_oe_output_32 (Windows)

`OE_SIGNALSTREAMPATH`*

Obtain path specifying hardware signal data stream

option value type	`char *`
option description	A character string specifying the signal stream path
default value	/dev/xillybus_oe_signal_8, \\.\xillybus_oe_signal_8 (Windows)

`OE_DEVICEMAP`

The device map.

option value type	`oe_device_t *`
option description	Pointer to a pre-allocated array of `oe_device_t` structs
default value	N/A

`OE_NUMDEVICES`

The number of devices in the device map.

option value type	`oe_reg_val_t`
option description	The number of devices supported by the firmware
default value	N/A

`OE_MAXREADFRAMESIZE`

The maximal size of a frame produced by a call to `oe_read_frame` in bytes. This number is the size of the frame produced by every device within the device map that generates read data.

option value type	`oe_reg_val_t`
option description	Maximal read frame size in bytes
default value	N/A

`OE_WRITEFRAMESIZE`

The maximal size of a frame accepted by a call to `oe_write_frame` in bytes. This number is the size of the frame provided to `oe_write_frame` to update all output devices synchronously.

option value type	`oe_reg_val_t`
option description	Maximal write frame size in bytes
default value	N/A

`OE_RUNNING`

Hardware acquisition run state. Any value greater than 0 indicates that acquisition is running.

option value type	`oe_reg_val_t`
option description	Any value greater than 0 will start acquisition
default value	False

`OE_SYSCLKHZ`

System clock frequency in Hz. Read frame clock values are are incremented at this rate.

option value type	`oe_reg_val_t`
option description	System clock frequency in Hz
default value	N/A

oe_set_opt

Set context options.

```c
int oe_set_opt(oe_ctx ctx, int option, const void* value, size_t size);
```

Arguments

- `ctx` context
- `option` option to set
- `value` value to set `option` to
- `size` length of `value` in bytes

Returns `int`

- 0: success
- Less than 0: `oe_error_t`

Description

The `oe_set_opt` function sets the option specified by the `option` argument to the value pointed to by the `value` argument within `ctx`. The `size` indicates the size of the `value` in bytes.

The following context options can be set:

`OE_CONFIGSTREAMPATH*`

Set path specifying configuration data stream.

option value type	`char *`
option description	A character string specifying the configuration stream path
default value	/dev/xillybus_oe_config_32, \\.\xillybus_oe_config_32 (Windows)

`OE_READSTREAMPATH*`

Set path specifying input data stream.

option value type	`char *`
option description	A character string specifying the input stream path
default value	/dev/xillybus_oe_input_32, \\.\xillybus_oe_input_32 (Windows)

`OE_WRITESTREAMPATH*`

Set path specifying input data stream.

option value type	`char *`
option description	A character string specifying the output stream path
default value	/dev/xillybus_oe_output_32, \\.\xillybus_oe_output_32 (Windows)

`OE_SIGNALSTREAMPATH*`

Set path specifying hardware signal data stream

option value type	`char *`
option description	A character string specifying the signal stream path
default value	/dev/xillybus_oe_signal_8, \\.\xillybus_oe_signal_8 (Windows)

`OE_RUNNING`**

Set/clear master clock gate. Any value greater than 0 will start acquisition. Writing 0 to this option will stop acquisition, but will not reset context options or the sample counter.

option value type	`oe_reg_val_t`
option description	Any value greater than 0 will start acquisition
default value	0

`OE_RESET`**

Trigger global hardware reset. Any value great than 0 will trigger a hardware reset. In this case, acquisition is stopped and all global hardware state (e.g. sample counters, etc) is defaulted.

option value type	`oe_reg_val_t`
option description	Any value greater than 0 will trigger a reset
default value	Untriggered

* Invalid following a successful call to `oe_init_ctx`. Before this, will return with error code `OE_EINVALSTATE`.

** Invalid until a successful call to `oe_init_ctx`. After this, will return with error code `OE_EINVALSTATE`.

oe_read_reg

Read a configuration register on a specific device.

```
int oe_read_reg(const oe_ctx ctx, size_t dev_idx, oe_reg_addr_t addr, oe_reg_val_t *value);
```

Arguments

- `ctx` context
- `dev_idx` physical index number
- `addr` The address of register to write to
- `value` pointer to an int that will store the value of the register at `addr` on `dev_idx`

Returns `int`

- 0: success
- Less than 0: `oe_error_t`

Description

`oe_read_reg` is used to read the value of configuration registers from devices within the current device map. This can be used to verify the success of calls to `oe_read_reg` or to obtain state information about devices managed by the current context.

oe_write_reg

Set a configuration register on a specific device.

```
int oe_write_reg(const oe_ctx ctx, size_t dev_idx, oe_reg_addr_t addr, oe_reg_val_t value);
```

Arguments

- `ctx` context
- `dev_idx` the device index to read from
- `addr` register address within the device specified by `dev_idx` to write to
- `value` value with which to set the register at `addr` on the device specified by `dev_idx`

Returns `int`

- 0: success
- Less than 0: `oe_error_t`

Description

`oe_write_reg` is used to write the value of configuration registers from devices within the current device map. This can be used to set configuraiton registers for devices managed by the current context. For example, this is used to perform configuration of ADCs that exist in a device map. Note that successful return from this function does not guarantee that the register has been properly set. Confirmation of the register value can be made using a call to `oe_read_reg`.

oe_read_frame

Read high-bandwidth input data stream.

```
int oe_read_frame(const oe_ctx ctx, oe_frame_t **frame)
```

Arguments

- `ctx` context
- `frame` Pointer to a `oe_frame_t` pointer

Returns `int`

- 0: success
- Less than 0: `oe_error_t`

Description

`oe_read_frame` allocates host memory and populates it with an `oe_frame_t` struct corresponding to a single frame, with a read header, from the data input channel. This call will block until either enough data to construct a frame is available on the data input stream or `oe_destroy_ctx` is called. It is the user's repsonisbility to free the resources allocated by this call by passing the resulting frame pointer to `oe_destroy_frame`.

oe_write_frame

Write a frame to the output data channel.

```
int oe_write_frame(const oe_ctx ctx, oe_frame_t *frame)
```

Arguments

- `ctx` context
- `frame` pointer to an `oe_frame_t`

Returns `int`

- 0: success
- Less than 0: `oe_error_t`

Description

`oe_write_frame` writes a pre-allocated and populated `stuct` corresponding to a single frame, with a write header, into the asynchronous data output channel from host memory. If the frame specifies that devices without write capabilities should be written to, this function will return `OE_EWRITEFAILURE`.

oe_destroy_frame

Free heap-allocated frame.

```
void oe_destroy_frame(oe_frame_t *frame);
```

Arguments

- `frame` pointer to an `oe_frame_t`

Returns `void`

There is no return value.

Description

`oe_destroy_frame` frees a heap-allocated frame. It is generally used to clean up the resources allocated by `oe_read_frame`.

oe_version

Report the oepcie library version.

```
void oe_version(int major, int minor, int patch)
```

Arguments

- `major` major library version
- `minor` minor library version
- `patch` patch number

Returns `void`

There is no return value.

Description

This library uses semantic versioning. Briefly, the major revision is for incompatible API changes. Minor version is for backwards compatible changes. The patch number is for backwards-compatible bug fixes.

oe_error_st

Convert an error number into a human readable string.

```
const char *oe_error_str(int err)
```

arguments

- `err` error code

returns `const char *`

Pointer to an error message string

oe_device_str

Convert a device ID into human readable string. *Note*: This is an extension function available in oedevices.h.

```
const char *oe_device(ind dev_id)
```

Arguments

- `dev_id` device id

Returns `const char *`

Pointer to a device id string