# Open SoC Debug Documentation Library

*Release 0.1*

The Open SoC Debug Contributors

Mar 30, 2020

# Documentation Parts

This is the reference implementation of the Open SoC Debug host software, i.e. the part of Open SoC Debug typically executed on a "standard" PC. The documentation is split in three parts.

The *Overview Documents* provide a general big-picture introduction to the ideas and concepts of OSD. They are written for a wide technical and non-technical audience.

The *Developer Documentation* provides techncial information about the implementation of the OSD host software. Read this documentation for information how to create software based on `libosd`, or how to extend the reference implementation itself.

Finally, the *(End) User Documentation* shows how to use the various tools provided by this reference implementation. This documentation is intended for end users, i.e. people trying to debug a system using OSD.

CHAPTER 1

# Introduction

This is the reference implementation of the Open SoC Debug host software, i.e. the part of Open SoC Debug typically executed on a "standard" PC.

## 1.1 About the code

A couple of quick facts before we start:

- Most code is written in object-oriented C.
- The code targets POSIX platforms (e.g. Linux). Windows is not supported right now (patches are welcome!).
- ZeroMQ is used heavily for communication between the individual components.
- All code is licensed under the permissive MIT license.

On a high level, the implementation is split into two parts:

- `libosd`, a reusable software library which does all the heavy lifting and provides an API for debug tools on the host. This code can be found in `src/libosd`.
- A set of tools which use libosd to fulfill a well-defined task (e.g. interface to a device, or log trace messages). The associated code can be found in `src/tools`.

## 1.2 The software architecture

The Open SoC Debug host software is composed of a set of standalone tools which each fulfill a single task. In many cases, a tool is identical to a debug module on the host.

The tools communicate via ZeroMQ sockets. These sockets provide an abstraction over the actual transport type. Typically, TCP is chosen (which allows for communication between modules on a single machine, but also on different machines). Also available are in-process communication via shared memory (useful if two tools are implemented inside a single process), socket communication (a bit faster than TCP when running on a single machine), and others.

Fig. 1.1: An overview of all OSD host software components from a user's perspective

Figure 1.1 presents an overview of the OSD host software architecture.

At the center of the architecture is the osd-host-controller. The host subnet controller has two jobs:

- It provides a central connection point for all tools and acts as message router.
- It provides low-level communication tasks, i.e. it assigns DI addresses to host debug modules.

From a user's perspective, all interesting functionality is implemented as host debug module. In many cases, each host debug module is implemented as standalone tool. For example, the tool osd-systrace-log is a host debug module which writes all STM messages it receives into a file. Other examples include a gdb server debug module which enables GDB clients to connect to an OSD-enabled system, or a trace logger, which writes instruction traces into a file.

Finally, the osd-device-gateway tool connects the host to a target device, e.g. an FPGA. It registers itself as gateway with the osd-host-controller, and transparently extends the debug interconnect to the subnet on the target device. The communication interface between the target device and the osd-device-gateway depends on whatever physical interfaces are available on the target (typically UART or USB). This device-host communication is encapsulated by GLIP, hence all GLIP-supported communication methods are also supported.

CHAPTER 2

# OSD Software Developer Documentation

## 2.1 OSD Host Communication Protocol

### 2.1.1 Message Types

All data on the host is transferred using ZeroMQ. The protocol used inside the ZeroMQ messages is described in this section.

All ZeroMQ messages consist of three frames.

Table 2.1: ZeroMQ message frame structure

| Frame | Name | Description |
|---|---|---|
| 1 | `src` | Message source (ZeroMQ identity frame) |
| 2 | `type` | Type of the message. Either `D` for data messages (encapsulated DI packets), or `M` for management messages (host only). |
| 3 | `payload` | Payload of the message. Its contents depend on the type. |

#### Data Messages

Data messages must have the `type` frame set to `D`. The `payload` field contains then a full OSD DI packet as an array of `uint16_t` words in system-native byte ordering (i.e. usually little endian).

#### Management Messages

Management messages must have the `type` frame set to `M`. These messages are used as a protocol layer below the DI, e.g. to dynamically aquire and release an address in the debug interconnect (which later can be used when sending data messages).

#### DIADDR_REQUEST

- Source: any host debug module
- Target: host subnet controller

Request a new (previously unused) Debug Interconnect address from the host controller for this subnet.

The subnet controller responds with a management message containing the assigned address as unsigned integer as payload. If the address assignment failed, a `NACK` message is sent instead.

### DIADDR_RELEASE

- Source: any host debug module
- Target: host subnet controller

Release the DI address assigned to the source.

If successsful, the subnet controller responds with an `ACK` message. If not successful, a `NACK` message is sent.

### GW_REGISTER <subnet-addr>

- Source: any
- Target: host subnet controller

Register the source of this message as gateway for all traffic intended for subnet <*subnet-addr*>. <*subnet-addr*> is given as decimal integer (base 10).

If successsful, the subnet controller responds with an `ACK` message. If not successful, a `NACK` message is sent.

### GW_UNREGISTER <subnet-addr>

- Source: any
- Target: host subnet controller

Remove the association of the gateway with subnet <*subnet-addr*>. <*subnet-addr*> is given as decimal integer (base 10).

If successsful, the subnet controller responds with an `ACK` message. If not successful, a `NACK` message is sent.

### ACK

- Source: any
- Target: any

Generic acknowledgement “operation successful”.

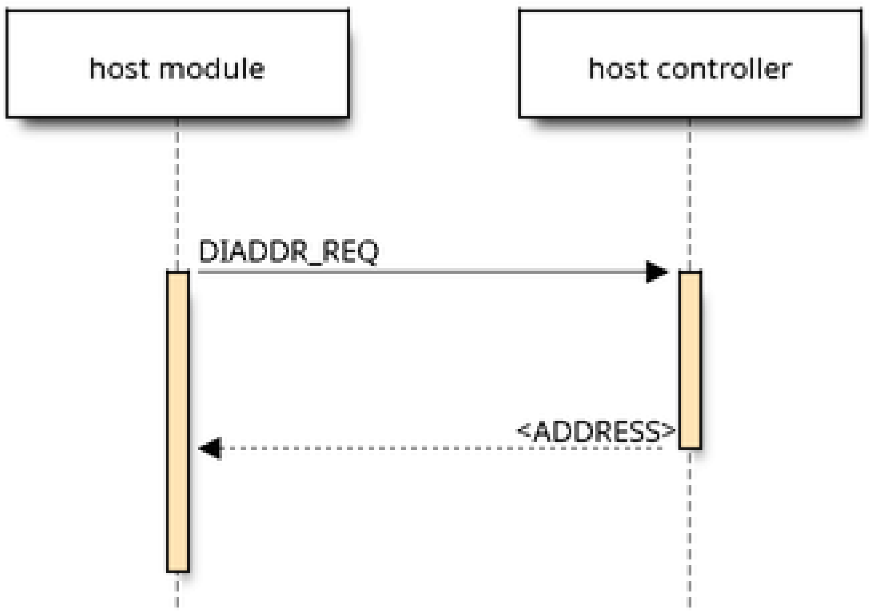
### NACK

- Source: any
- Target: any

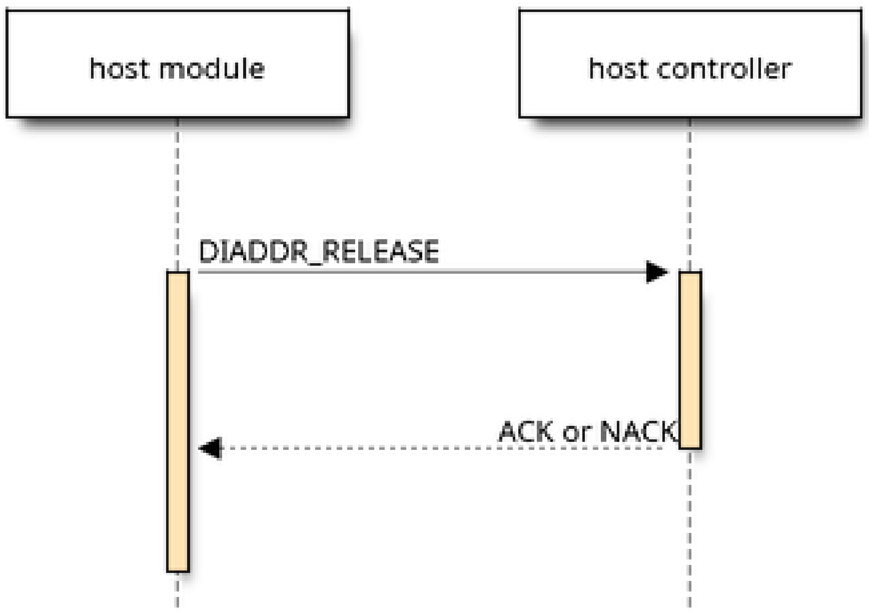
Generic error message “operation failed”.

### 2.1.2 Protocol Flows

#### Host Debug Module: Connect



#### Host: Disconnect



#### Host: Register as Gateway Device

## 2.2 Tutorial: How to create a new debug tool

**Todo:** missing content

## 2.3 Unit Tests

The OSD software comes with an extensive set of unit tests, written using the Check framework[1].

You can run the unit tests by calling `make check` in the `build` directory.

```
make check
```

Check writes two types of log files: one log summary `tests/unit/test-suite.log` and an individual log file per test named after the test, e.g. `tests/unit/check_hostmod.log`. `printf()` output from the individual tests is only written to the individual log files if the tests pass. To get pass/fail remarks per test function set the `CK_VERBOSITY` environment variable to `verbose` when running a test.

**Note:** If you don't find expected `printf()` output in the test log file try adding a `fflush(stdout)` statement after the `printf()`.

```
# view test summary
less tests/unit/test-suite.log

# obtain and view verbose logs for check_hostmod
CK_VERBOSITY=verbose make check
less tests/unit/check_hostmod.log
```

Sometimes it's helpful to debug the tests themselves using gdb. The following command line runs a compiled test under gdb (replace `check_hostmod` with your test):

```
make check # build the tests (if not already done)
CK_FORK=no libtool --mode=execute gdb tests/unit/check_hostmod
```

To further control the execution of tests the check framework offers a number of environment variables. These are listed in the documentation[2].

### 2.3.1 Analyzing Code Coverage

To gain insight into which parts of the software are tested you can generate a coverage report. To build and execute all tests and to finally generate a coverage report as HTML page run

```
make check-code-coverage
firefox tests/unit/opensocdebug-*-coverage/index.html
```

[1] https://libcheck.github.io/check/
[2] https://libcheck.github.io/check/doc/check_html/check_7.html

### 2.3.2 Find memory leaks with Valgrind

To check the code for memory errors you can run the unit test suite under Valgrind with its memcheck[3] tool. Valgrind support must be enabled at configure time by passing the options `--enable-valgrind --disable-asan` to the `configure` script.

```
# run all tests with Valgrind
make check-valgrind
```

### 2.3.3 Find memory errors with Address Sanitizer (ASan)

Address Sanitizer[4] (ASan) is a tool to find common classes of memory errors, such as out of bounds accesses. It complements Valgrind especially for data on the stack.

To enable ASan run `configure` with `--disable-valgrind --enable-asan`. (It's not possible to create a build with ASan and Valgrind enabled at the same time.) ASan is then built-in and enabled in all produced binaries, including the unit tests. In case of errors ASan will report them during the program run.

```
# ASan is automatically enabled when running the test suite
make check
```

## 2.4 Debugging

### 2.4.1 Run the program with GDB

Before *make install* is called all compiled binaries in the source tree are wrapped by libtool. To run GDB on them, use the following command line:

```
# general
libtool --mode=execute gdb --args YOUR_TOOL

# for example
libtool --mode=execute gdb --args ./src/tools/osd-target-run/osd-target-run -e ~/
↪src/baremetal-apps/hello/hello.elf -vvv
```

### 2.4.2 GDB helpers

GDB can call functions in the program binary when the program is stopped (e.g. when a breakpoint hit). This can be used to dump useful information from data structures in a readable form.

```
# dump a DI packet
(gdb) p osd_packet_dump(pkg, stdout)
Packet of 5 data words:
DEST = 4096, SRC = 1, TYPE = 2 (OSD_PACKET_TYPE_EVENT), TYPE_SUB = 0
Packet data (including header):
  0x1000
  0x0001
  0x8000
  0x0000
  0x0000
$5 = void
```

[3] https://valgrind.org/docs/manual/mc-manual.html
[4] https://github.com/google/sanitizers/wiki/AddressSanitizer

### 2.4.3 Profiling

Generating an execution profile of the OSD tools can be helpful to reduce the CPU consumption, increase throughput, etc. To generate a profile we recommend the Linux `perf` tool. (gprof doesn't play nice with ZeroMQ.)

#### Setup perf

Install perf using the package manager of your distribution.

```
# install perf
# for Ubuntu/Debian
sudo apt-get install perf
# for SUSE
sudo zypper install perf
```

Running perf as non-root user requires a couple kernel settings to be changed.

```
# Give access to kernel pointers
sudo sh -c " echo 0 > /proc/sys/kernel/kptr_restrict"

# Give access to perf events
sudo sh -c " echo 0 > /proc/sys/kernel/perf_event_paranoid"

# Make kernel permissions permanent after the next reboot (optional)
echo "kernel.kptr_restrict=0\nkernel.perf_event_paranoid=0" >> /etc/sysctl.conf"
```

#### Profile application

First, build the application as usual. Then, from inside the source tree, run the tool with `perf record` to collect an execution profile.

```
# example: profile osd-target-run (without call stacks)
libtool --mode=execute perf record ./src/tools/osd-target-run/osd-target-run -e ~/
↪src/baremetal-apps/hello/hello.elf --systrace -vvv

# the same with call stacks
libtool --mode=execute perf record -g ./src/tools/osd-target-run/osd-target-run -e␣
↪~/src/baremetal-apps/hello/hello.elf --systrace -vvv
```

#### Evaluate profile

`perf record` stores the recorded events in a file called `perf.data` in the directory it was called in. To display this data nicely run `perf report`.

```
perf report
```

## 2.5 API Documentation

`libosd` consists of a number of classes, each handling one distict task. Each class is contained in one C header file.

### 2.5.1 osd_hostmod class

This class represents a debug module on the host.

`osd_hostmod` implements functionality common to all host debug modules, and provides extension for individual host module implementations. Most importantly, `osd_hostmod` encapsulates all communication with the host controller.

### Usage

```
#include <osd/osd.h>
#include <osd/hostmod.h>
```

### Example

```
#include <osd/osd.h>
#include <osd/hostmod.h>

const char* HOST_CONTROLLER_URL = "tcp://localhost:6666";

// initialize class
osd_hostmod_ctx *hostmod_ctx;
osd_hostmod_new(&hostmod_ctx);

// connect to host controller
osd_hostmod_connect(hostmod_ctx, HOST_CONTROLLER_URL);

// the subnet controller assigns this host module a unique address
uint16_t addr;
addr = osd_hostmod_get_diaddr(hostmod_ctx);
printf("This module got the address %d assigned.\n", addr);

// read register 0x0000 from module with address 0x0000
uint16_t result;
osd_hostmod_reg_read(hostmod_ctx, 0, 0, 16, &result, 0);
printf("Read returned value %u.\n", result);

// disconnect from host controller
osd_hostmod_disconnect(hostmod_ctx);

// cleanup
osd_hostmod_free(&hostmod_ctx);
```

### Public Interface

### Defines

**OSD_HOSTMOD_BLOCKING**
Flag: fully blocking operation (i.e. wait forever)

### Typedefs

**typedef** *osd_result* **(* osd_hostmod_event_handler_fn)** (void *, struct *osd_packet* *)
Event handler function prototype

The ownership of packet is passed to the handler function.

### Functions

*osd_result* **osd_hostmod_new** (struct osd_hostmod_ctx ** *ctx*, struct osd_log_ctx * *log_ctx*, const char * *host_controller_address*, *osd_hostmod_event_handler_fn* *event_handler*, void * *event_handler_arg*)

Create new osd_hostmod instance

**Return** OSD_OK on success, any other value indicates an error

**See** *osd_hostmod_free()*

**Parameters**

- `[out] ctx`: the osd_hostmod_ctx context to be created
- `[in] log_ctx`: the log context to be used. Set to NULL to disable logging
- `[in] host_controller_address`: ZeroMQ endpoint of the host controller
- `[in] event_handler`: function called when a new event packet is received
- `[in] event_handler_arg`: argument passed to the event handler callback

void **osd_hostmod_free** (struct osd_hostmod_ctx ** *ctx*)

Free and NULL a communication API context object

Call *osd_hostmod_disconnect()* before calling this function.

**Parameters**

- `ctx`: the osd_com context object

*osd_result* **osd_hostmod_connect** (struct osd_hostmod_ctx * *ctx*)

Connect to the host controller

**Return** OSD_OK on success, any other value indicates an error

**See** *osd_hostmod_disconnect()*

**Parameters**

- `ctx`: the osd_hostmod_ctx context object

*osd_result* **osd_hostmod_disconnect** (struct osd_hostmod_ctx * *ctx*)

Shut down all communication with the device

**Return** OSD_OK on success, any other value indicates an error

**See** osd_hostmod_run()

**Parameters**

- `ctx`: the osd_hostmod context object

bool **osd_hostmod_is_connected** (struct osd_hostmod_ctx * *ctx*)

Is the connection to the device active?

**Return** 1 if connected, 0 if not connected

**See** *osd_hostmod_connect()*

**See** *osd_hostmod_disconnect()*

**Parameters**

- `ctx`: the osd_hostmod context object

*osd_result* **osd_hostmod_reg_read** (struct osd_hostmod_ctx * *ctx*, void * *reg_val*, uint16_t *diaddr*, uint16_t *reg_addr*, int *reg_size_bit*, int *flags*)

Read a register of a module in the debug system

Unless the flag OSD_HOSTMOD_BLOCKING has been set this function times out if the module does not reply within a ZMQ_RCV_TIMEOUT milliseconds.

**Return** OSD_OK on success, any other value indicates an error

**Return** OSD_ERROR_TIMEDOUT if the register read timed out (only if OSD_HOSTMOD_BLOCKING is not set)

**See** osd_hostmod_write()

**Parameters**

- `ctx`: the osd_hostmod_ctx context object
- `[out] reg_val`: the result of the register read. Preallocate a variable large enough to hold `reg_size_bit` bits.
- `diaddr`: the DI address of the module to read the register from
- `reg_addr`: the address of the register to read
- `reg_size_bit`: size of the register in bit. Supported values: 16, 32, 64 and 128.
- `flags`: flags. Set OSD_HOSTMOD_BLOCKING to block indefinitely until the access succeeds.

*osd_result* **osd_hostmod_reg_write** (struct osd_hostmod_ctx * *ctx*, const void * *reg_val*, uint16_t *diaddr*, uint16_t *reg_addr*, int *reg_size_bit*, int *flags*)

Write a register of a module in the debug system

**Return** OSD_OK on success, any other value indicates an error

**Return** OSD_ERROR_TIMEDOUT if the register read timed out (only if OSD_HOSTMOD_BLOCKING is not set)

**Parameters**

- `ctx`: the osd_hostmod_ctx context object
- `reg_val`: the data to be written. Provide enough data according to `reg_size_bit`
- `diaddr`: the DI address of the accessed module
- `reg_addr`: the address of the register to read
- `reg_size_bit`: size of the register in bit. Supported values: 16, 32, 64 and 128.
- `flags`: flags. Set OSD_HOSTMOD_BLOCKING to block indefinitely until the access succeeds.

*osd_result* **osd_hostmod_reg_setbit** (struct osd_hostmod_ctx * *hostmod_ctx*, unsigned int *bitnum*, bool *bitval*, uint16_t *diaddr*, uint16_t *reg_addr*, int *reg_size_bit*, int *flags*)

Set (or unset) a bit in a debug module configuration register

The setting of a single bit requires a read-modify-write cycle of the register, which is not atomic and not locked.

**Return** OSD_OK on success, any other value indicates an error

**Return** OSD_ERROR_TIMEDOUT if the register read timed out (only if OSD_HOSTMOD_BLOCKING is not set)

**Parameters**

- `ctx`: the osd_hostmod_ctx context object

- `bitnum`: bit to modify (bit 0 = LSB)
- `bitval`: value to set the bit to
- `diaddr`: the DI address of the accessed module
- `reg_addr`: the address of the register to read
- `reg_size_bit`: size of the register in bit. Supported values: 16, 32, 64 and 128.
- `flags`: flags. Set OSD_HOSTMOD_BLOCKING to block indefinitely until the access succeeds.

uint16_t **`osd_hostmod_get_diaddr`** (struct osd_hostmod_ctx * *ctx*)

Get the DI address assigned to this host debug module

The address is assigned during the connection, i.e. you need to call *osd_hostmod_connect()* before calling this function.

**Return** the address assigned to this debug module

**Parameters**

- `ctx`: the osd_hostmod_ctx context object

unsigned int **`osd_hostmod_get_max_event_words`** (struct osd_hostmod_ctx * *ctx*, unsigned int *di_addr_target*)

Get the maximum number paylaod of words in an event packet

**Return** the number of payload words in an event packet sent to `di_addr_target`

**Parameters**

- `ctx`: the osd_hostmod_ctx context object

*osd_result* **`osd_hostmod_event_send`** (struct osd_hostmod_ctx * *ctx*, const struct *osd_packet* * *event_pkg*)

Send an event packet to its destination

**Return** OSD_OK on success, any other value indicates an error

**Parameters**

- `ctx`: the osd_hostmod_ctx context object
- `event_pkg`: the event packet to be sent

*osd_result* **`osd_hostmod_event_receive`** (struct osd_hostmod_ctx * *ctx*, struct *osd_packet* ** *event_pkg*, int *flags*)

Receive an event packet

By default, this function times out with OSD_ERROR_TIMEOUT if no packet was received. Pass OSD_HOSTMOD_BLOCKING to `flags` to make the function block until a packet is received.

**Return** OSD_OK on success, any other value indicates an error

**Parameters**

- `ctx`: the osd_hostmod_ctx context object
- `[out] event_pkg`: the received event packet. Allocated by this function, must be free'd by caller after use.
- `flags`: a ORed list of flags (see description)

*osd_result* **`osd_hostmod_get_modules`** (struct osd_hostmod_ctx * *ctx*, unsigned int *subnet_addr*, struct osd_module_desc ** *modules*, size_t * *modules_len*)

Get a list of all debug modules in a given subnet

**Return** OSD_OK on success OSD_ERROR_RESULT_PARTIAL if at least one module failed to enumerate any other value indicates an error

**Parameters**

- `ctx`: the osd_hostmod_ctx context object
- `subnet_addr`: the address of the subnet to query for modules
- `[out] modules`: the modules found in the given subnet. The array is allocated by this function and ownership passed to the caller, who must free it after use.
- `[out] module_len`: the number of entries in `modules`

*osd_result* **`osd_hostmod_mod_describe`** (struct osd_hostmod_ctx * *ctx*, uint16_t *di_addr*, struct osd_module_desc * *desc*)

Get the description fields of a debug module (type, vendor, version)

**Return** OSD_OK on success, any other value indicates an error

**Parameters**

- `ctx`: the osd_hostmod_ctx context object
- `di_addr`: the DI address of the module to describe
- `[out] desc`: a struct describing the module

*osd_result* **`osd_hostmod_mod_set_event_dest`** (struct osd_hostmod_ctx * *ctx*, uint16_t *di_addr*, int *flags*)

Set this host module as event destination for the module at `di_addr`

By default, this function times out with OSD_ERROR_TIMEOUT if no packet was received. Pass OSD_HOSTMOD_BLOCKING to `flags` to make the function block until a packet is received.

**Return** OSD_OK on success, any other value indicates an error

**See** *osd_hostmod_mod_set_event_active()*

**Parameters**

- `ctx`: the osd_hostmod_ctx context object
- `di_addr`: the address of the DI module
- `flags`: a ORed list of flags (see description)

*osd_result* **`osd_hostmod_mod_set_event_active`** (struct osd_hostmod_ctx * *ctx*, uint16_t *di_addr*, bool *enabled*, int *flags*)

Enable/disable sending of events by the module at DI address `di_addr`

Events are sent to the DI address configured in the module. Use osd_hostmod_mod_set_evdest() to make this host module receive the events.

By default, this function times out with OSD_ERROR_TIMEOUT if no packet was received. Pass OSD_HOSTMOD_BLOCKING to `flags` to make the function block until a packet is received.

**Return** OSD_OK on success, any other value indicates an error

**See** osd_hostmod_mod_set_evdest()

**Parameters**

- `ctx`: the osd_hostmod_ctx context object
- `di_addr`: the address of the DI module to enable/disable
- `enabled`: 1 to enable event sending, 0 to disable it
- `flags`: a ORed list of flags (see description)

struct osd_log_ctx* **osd_hostmod_log_ctx** (struct osd_hostmod_ctx * *ctx*)
: Get the logging context for this host module (internal use only)

### Internal Architecture

**Note:** This section is targeting developers working on the `osd_hostmod` module.

**Note:** TBD

## 2.5.2 osd_hostctrl class

This class contains the host controller. The host controller is the central instance in a host subnet and has the following tasks:

- It serves as central connection point for all host modules (i.e. forming a star topology). As such, it handles the setup and teardown of connections, as well as assigning of DI addreses.
- It routes messages between host modules.
- Gateways can connect to the host controller to extend the debug network beyond the host.

### Usage

```
#include <osd/osd.h>
#include <osd/hostctrl.h>
```

### Public Interface

### Functions

*osd_result* **osd_hostctrl_new** (struct osd_hostctrl_ctx ** *ctx*, struct osd_log_ctx * *log_ctx*, const char * *router_address*)
: Create new host controller

The host controller will listen to requests at `router_addres`

**Return** OSD_OK if initialization was successful, any other return code indicates an error

**Parameters**

- `ctx`: context object
- `log_ctx`: logging context
- `router_address`: ZeroMQ endpoint/URL the host controller will listen on

*osd_result* **osd_hostctrl_start** (struct osd_hostctrl_ctx * *ctx*)
: Start host controller

*osd_result* **osd_hostctrl_stop** (struct osd_hostctrl_ctx * *ctx*)
: Stop host controller

void **osd_hostctrl_free** (struct osd_hostctrl_ctx ** *ctx_p*)
: Free the host controller context object (destructor)

By calling this function all resources associated with the context object are freed and the ctx_p itself is NULLed.

**Parameters**

- `ctx_p`: the host controller context object

bool **`osd_hostctrl_is_running`** (struct osd_hostctrl_ctx * *ctx*)
Is the host controller running?

**Return** true if the host controller is running, false otherwise

**Parameters**

- `ctx`: the context object

### 2.5.3 osd_gateway class

This class implements a gateway between two debug subnets, typically between the subnet on the host and the one on a "target device", an OSD enabled chip (as an ASIC, on an FPGA or running in simulation). On the host side, the gateway registers with the host controller for a given subnet. It then receives all traffic going to this subnet. On the device side, the OSD packets are transformed into Data Transport Datagrams (length-value encoded OSD packets). Standard read()/write() callback function can then be implemented to perform the actual data transfer to the device, possibly using another suitable library (such as GLIP).

#### Behavior on connection loss

The connected device can drop the connection at any time. That is signaled by the device by returning OSD_ERROR_NOT_CONNECTED from its packet_read() and packet_write() callback functions. When osd_gateway detects this kind of loss in connectivity, it disconnects the gateway from the host controller.

#### Usage

```
#include <osd/osd.h>
#include <osd/gateway.h>
```

#### Public Interface

#### Typedefs

**`typedef`** *osd_result* **`(* packet_read_fn)`** (struct *osd_packet* ***pkg*, void **cb_arg*)
Read a *osd_packet* from the device

**Return** OSD_ERROR_NOT_CONNECTED if the not connected to the device

**Return** OSD_OK if successful

**Parameters**

- `pkg`: the packet to read to (allocated by the called function)
- `cb_arg`: an user-defined callback argument

**`typedef`** *osd_result* **`(* packet_write_fn)`** (const struct *osd_packet* **pkg*, void **cb_arg*)
Write a *osd_packet* to the device

**Return** OSD_ERROR_NOT_CONNECTED if the not connected to the device

**Return** OSD_OK if successful

**Parameters**

- `pkg`: the packet to write
- `cb_arg`: an user-defined callback argument

### Functions

*osd_result* **`osd_gateway_new`** (struct osd_gateway_ctx ** *ctx*, struct osd_log_ctx * *log_ctx*, const char * *host_controller_address*, uint16_t *device_subnet_addr*, *packet_read_fn* *packet_read*, *packet_write_fn* *packet_write*, void * *cb_arg*)

Create new osd_gateway instance

**Return** OSD_OK on success, any other value indicates an error

**See** *osd_gateway_free()*

**Parameters**

- `[out] ctx`: the osd_gateway_ctx context to be created
- `[in] log_ctx`: the log context to be used. Set to NULL to disable logging
- `[in] host_controller_address`: ZeroMQ endpoint of the host controller
- `[in] device_subnet_addr`: Subnet address of the device
- `[in] packet_read`: callback function to perform a read from the device
- `[in] packet_write`: callback function to perform a write to the device
- `[in] cb_arg`: an user-defined pointer passed to all callback functions, such as packet_read and packet_write. Can be used to pass context objects to the callbacks. Set to NULL if unused. Note: the callbacks are called from different threads, make sure all uses of cb_arg inside the callbacks are thread safe.

void **`osd_gateway_free`** (struct osd_gateway_ctx ** *ctx*)

Free and NULL a communication API context object

Call *osd_gateway_disconnect()* before calling this function.

**Parameters**

- `ctx`: the osd_com context object

*osd_result* **`osd_gateway_connect`** (struct osd_gateway_ctx * *ctx*)

Connect to the device and to the host controller

**Return** OSD_OK on success, any other value indicates an error

**See** *osd_gateway_disconnect()*

**Parameters**

- `ctx`: the osd_gateway_ctx context object

*osd_result* **`osd_gateway_disconnect`** (struct osd_gateway_ctx * *ctx*)

Shut down all communication with the device and the host controller

**Return** OSD_OK on success, any other value indicates an error

**See** osd_gateway_run()

**Parameters**

- `ctx`: the osd_hostmod context object

bool **`osd_gateway_is_connected`**(struct osd_gateway_ctx * *ctx*)

Is the connection to the device and to the host controller active?

**Return** 1 if connected, 0 if not connected

**See** *osd_gateway_connect()*

**See** *osd_gateway_disconnect()*

**Parameters**

- `ctx`: the context object

struct *osd_gateway_transfer_stats** **`osd_gateway_get_transfer_stats`**(struct osd_gateway_ctx * *ctx*)

Get statistics about the data transferred through the gateway

**`struct osd_gateway_transfer_stats`**

*#include <gateway.h>* Data transfer statistics

**See** *osd_gateway_get_transfer_stats()*

#### Public Members

struct timespec **`connect_time`**

uint64_t **`bytes_from_device`**

uint64_t **`bytes_to_device`**

## 2.5.4 osd_cl_mam class

A client for the Memory Access Module (MAM).

`cl_mam` is the counterpart to the Memory Access Module (MAM): it implements the MAM protocol to read and write data from and to memories.

### Usage

```
#include <osd/osd.h>
#include <osd/cl_mam.h>
```

### Public Interface

### Functions

*osd_result* **`osd_cl_mam_get_mem_desc`**(struct osd_hostmod_ctx * *hostmod_ctx*, unsigned int *mam_di_addr*, struct *osd_mem_desc* * *mem_desc*)

Obtain information about the memory connected to a MAM module

**Return** OSD_OK on success, any other value indicates an error

**Parameters**

- `hostmod_ctx`: the host module handling the communication
- `mam_di_addr`: DI address of the MAM module to get describe
- `[out] mem_desc`: pre-allocated memory descriptor for the result

*osd_result* **osd_cl_mam_write** (const struct *osd_mem_desc* * *mem_desc*, struct osd_hostmod_ctx * *hostmod_ctx*, const void * *data*, size_t *nbyte*, uint64_t *start_addr*)

Write data to the memory attached to a Memory Access Module (MAM)

The data does *not* need to be word-aligned. Writes across memory region boundaries are not allowed. This function blocks until the write is acknowledged by the memory.

**Return** OSD_OK if the write was successful any other value indicates an error

**See** *osd_cl_mam_read()*

**Parameters**

- `mem_desc`: descriptor of the target memory
- `hostmod_ctx`: the host module handling the communication
- `data`: the data to be written
- `nbyte`: the number of bytes to write
- `start_addr`: first byte address to write data to. All subsequent words are written to consecutive addresses.

*osd_result* **osd_cl_mam_read** (const struct *osd_mem_desc* * *mem_desc*, struct osd_hostmod_ctx * *hostmod_ctx*, void * *data*, size_t *nbyte*, uint64_t *start_addr*)

Read data from a memory attached to a Memory Access Module (MAM)

The data does *not* need to be word-aligned. Reads across memory region boundaries are not allowed. This function blocks until the write is acknowledged by the memory.

**Return** OSD_OK if the read was successful any other value indicates an error

**See** *osd_cl_mam_write()*

**Parameters**

- `mem_desc`: descriptor of the target memory
- `hostmod_ctx`: the host module handling the communication
- `data`: the returned read data. Must be preallocated and large enough for nbyte bytes of data.
- `nbyte`: the number of bytes to read
- `start_addr`: first byte address to read from

**struct osd_mem_desc_region**

*#include <cl_mam.h>* Information about a memory region

### Public Members

uint64_t **baseaddr**

uint64_t **memsize**

**struct osd_mem_desc**

*#include <cl_mam.h>* Information about a memory attached to a MAM module

### Public Members

unsigned int **di_addr**

DI address of the memory.

uint16_t **data_width_bit**

Data width in bit.

uint16_t **`addr_width_bit`**
Address width in bit.

uint8_t **`num_regions`**
Number of accessible memory regions.

**`struct osd_mem_desc_region osd_mem_desc::regions[8]`**
Memory region information.

### 2.5.5 osd_cl_scm class

API to access the functionality of the Subnet Control Module (SCM).

#### Usage

```
#include <osd/osd.h>
#include <osd/cl_scm.h>
```

#### Public Interface

#### Functions

*osd_result* **`osd_cl_scm_cpus_start`** (struct osd_hostmod_ctx * *hostmod_ctx*, unsigned int *subnet_addr*)
Start (un-halt) all CPUs in the SCM subnet

**See** *osd_cl_scm_cpus_stop()*

*osd_result* **`osd_cl_scm_cpus_stop`** (struct osd_hostmod_ctx * *hostmod_ctx*, unsigned int *subnet_addr*)
Stop (halt) all CPUs in the SCM subnet

*osd_result* **`osd_cl_scm_get_subnetinfo`** (struct osd_hostmod_ctx * *hostmod_ctx*, unsigned int *subnet_addr*, struct *osd_subnet_desc* * *subnet_desc*)
Get a description of a given subnet from the SCM

Read the system information from the device, as stored in the SCM

**`struct osd_subnet_desc`**

#### Public Members

uint16_t **`vendor_id`**

uint16_t **`device_id`**

uint16_t **`max_pkt_len`**

### 2.5.6 osd_cl_stm class

API to access the functionality of the System Trace Module (STM).

#### Usage

```
#include <osd/osd.h>
#include <osd/cl_stm.h>
```

### Public Interface

#### Typedefs

**typedef** void**(* osd_cl_stm_handler_fn)**(void *, const struct *osd_stm_desc* *, const struct *osd_stm_event* *)

#### Functions

*osd_result* **osd_cl_stm_get_desc**(struct osd_hostmod_ctx * *hostmod_ctx*, unsigned int *stm_di_addr*, struct *osd_stm_desc* * *stm_desc*)

Populate the STM descriptor with data from the debug module

**Return** OSD_OK on success OSD_ERROR_WRONG_MODULE if the module at stm_di_addr is not a STM any other value indicates an error

**Parameters**

- `hostmod_ctx`: the host module handling the communication
- `stm_di_addr`: DI address of the STM module to get describe
- `[out] stm_desc`: pre-allocated memory descriptor for the result

*osd_result* **osd_cl_stm_handle_event**(void * *arg*, struct *osd_packet* * *pkg*)

Event handler to process STM event, to be passed to a hostmod instance

bool **osd_cl_stm_is_print_event**(const struct *osd_stm_event* * *ev*)

Is the given STM event a sysprint event?

*osd_result* **osd_cl_stm_print_buf_new**(struct *osd_cl_stm_print_buf* ** *print_buf_p*)

Allocate memory for a new *osd_cl_stm_print_buf* struct

void **osd_cl_stm_print_buf_free**(struct *osd_cl_stm_print_buf* ** *print_buf_p*)

Free a *osd_cl_stm_print_buf* struct

*osd_result* **osd_cl_stm_add_to_print_buf**(const struct *osd_stm_event* * *ev*, struct *osd_cl_stm_print_buf* * *buf*, bool * *should_flush*)

Add a STM event to the print buffer

**struct osd_stm_desc**

*#include <cl_stm.h>* Information a System Trace Module

#### Public Members

unsigned int **di_addr**

DI address of the memory.

uint16_t **value_width_bit**

Size of the value in a trace event in bit.

**struct osd_stm_event**

*#include <cl_stm.h>* A single event emitted by the STM module

In case of an overflow in the system an overflow even is generated, containing the number of missed events. If the overflow value is set to a non-zero value, id and value are invalid.

#### Public Members

uint32_t **timestamp**

timestamp

uint16_t **id**
: event identifier

uint64_t **value**
: traced value

uint16_t **overflow**
: Number of lost packets due to overflow.

**struct osd_stm_event_handler**

### Public Members

const struct *osd_stm_desc** **stm_desc**

*osd_cl_stm_handler_fn* **cb_fn**

void* **cb_arg**

**struct osd_cl_stm_print_buf**

### Public Members

char* **buf**
: data buffer

size_t **len_buf**
: allocated size of |buf|

size_t **len_str**
: length of the string inside |buf|, excluding the zero termination

## 2.5.7 osd_cl_ctm class

API to access the functionality of the Core Trace Module (CTM).

### Usage

```
#include <osd/osd.h>
#include <osd/cl_ctm.h>
```

### Public Interface

### Typedefs

**typedef** void (***osd_cl_ctm_handler_fn)** (void *, const struct *osd_ctm_desc* *, const struct *osd_ctm_event* *)

### Functions

*osd_result* **osd_cl_ctm_get_desc** (struct osd_hostmod_ctx * *hostmod_ctx*, unsigned int *ctm_di_addr*, struct *osd_ctm_desc* * *ctm_desc*)
Populate the CTM descriptor with data from the debug module

**Return** OSD_OK on success OSD_ERROR_WRONG_MODULE if the module at ctm_di_addr is not a CTM any other value indicates an error

**Parameters**

- `hostmod_ctx`: the host module handling the communication
- `ctm_di_addr`: DI address of the CTM module to get describe
- `[out] ctm_desc`: pre-allocated memory descriptor for the result

*osd_result* **`osd_cl_ctm_handle_event`** (void * *arg*, struct *osd_packet* * *pkg*)
Event handler to process CTM event, to be passed to a hostmod instance

**`struct osd_ctm_desc`**
*#include <cl_ctm.h>* Information a Core Trace Module

## Public Members

unsigned int **`di_addr`**
DI address of the memory.

uint16_t **`addr_width_bit`**
Width of an address in bit.

uint16_t **`data_width_bit`**
Width of a data word in bit.

**`struct osd_ctm_event`**
*#include <cl_ctm.h>* A single event emitted by the CTM module

In case of an overflow in the system an overflow even is generated, containing the number of missed events. If the overflow value is set to a non-zero value, all other fields except for the timestamp are invalid.

## Public Members

uint16_t **`overflow`**
number of overflowed packets

uint32_t **`timestamp`**
timestamp

uint64_t **`npc`**
npc

uint64_t **`pc`**
pc

uint8_t **`mode`**
privilege mode

bool **`is_ret`**
executed instruction is a function return

bool **`is_call`**
executed instruction is a function call

bool **`is_modechange`**
executed instruction changed the privilege mode

**`struct osd_ctm_event_handler`**

## Public Members

const struct *osd_ctm_desc** **`ctm_desc`**

*osd_cl_ctm_handler_fn* **`cb_fn`**

void* **`cb_arg`**

### 2.5.8 osd_cl_cdm class

API to access the functionality of the Core Debug Module (CDM).

#### Usage

```
#include <osd/osd.h>
#include <osd/cl_cdm.h>
```

#### Public Interface

#### Typedefs

**typedef** void **(* osd_cl_cdm_handler_fn)** (void *, const struct *osd_cdm_desc* *, const struct *osd_cdm_event* *)

#### Functions

*osd_result* **osd_cl_cdm_get_desc** (struct osd_hostmod_ctx * *hostmod_ctx*, unsigned int *cdm_di_addr*, struct *osd_cdm_desc* * *cdm_desc*)
Populate the CDM descriptor with data from the debug module

**Return** OSD_OK on success OSD_ERROR_WRONG_MODULE if the module at cdm_di_addr is not a CDM any other value indicates an error

**Parameters**

- `hostmod_ctx`: the host module handling the communication
- `cdm_di_addr`: DI address of the CDM module to get describe
- `[out] cdm_desc`: pre-allocated memory descriptor for the result

*osd_result* **osd_cl_cdm_handle_event** (void * *arg*, struct *osd_packet* * *pkg*)
Event handler to process CDM event, to be passed to a hostmod instance

*osd_result* **cl_cdm_cpureg_read** (struct osd_hostmod_ctx * *hostmod_ctx*, struct *osd_cdm_desc* * *cdm_desc*, void * *reg_val*, uint16_t *reg_addr*, int *flags*)
Read data from an SPR of the CPU attached to a Core Debug Module (CDM)

**Return** OSD_OK if the read was successful any other value indicates an error

**See** osd_cl_cdm_cpureg_write()

**Parameters**

- `hostmod_ctx`: the host module handling the communication
- `[out] cdm_desc`: pre-allocated memory descriptor for the result
- `reg_val`: the returned read data
- `reg_addr`: address to read from
- `flags`: flags. Set OSD_HOSTMOD_BLOCKING to block indefinitely until the access succeeds.

*osd_result* **cl_cdm_cpureg_write** (struct osd_hostmod_ctx * *hostmod_ctx*, struct *osd_cdm_desc* * *cdm_desc*, const void * *reg_val*, uint16_t *reg_addr*, int *flags*)
Write data to an SPR of the CPU attached to a Core Debug Module (CDM)

**Return** OSD_OK if the write was successful any other value indicates an error

**See** osd_cl_cdm_cpureg_read()

**Parameters**

- `hostmod_ctx`: the host module handling the communication
- `[out] cdm_desc`: pre-allocated memory descriptor for the result
- `reg_val`: the data to be written
- `reg_addr`: address to write data to
- `flags`: flags. Set OSD_HOSTMOD_BLOCKING to block indefinitely until the access succeeds.

**`struct osd_cdm_desc`**
*#include <cl_cdm.h>* Information about a Core Debug Module

### Public Members

unsigned int **`di_addr`**
DI address of CDM module.

uint16_t **`core_ctrl`**
Control signal for CPU core.

uint16_t **`core_reg_upper`**
Most significant bits of the SPR address.

uint16_t **`core_data_width`**
CPU data width in bits.

**`struct osd_cdm_event`**
*#include <cl_cdm.h>* A single event generated by the CDM module

### Public Members

bool **`stall`**
indicates the debugger that the CPU core is stalled.

**`struct osd_cdm_event_handler`**

### Public Members

const struct *osd_cdm_desc** **`cdm_desc`**

*osd_cl_cdm_handler_fn* **`cb_fn`**

void* **`cb_arg`**

## 2.5.9 osd_log class

Common logging functionality.

### Usage

```
#include <osd/osd.h>
```

## Public Interface

**typedef** void **(* osd_log_fn)** (struct osd_log_ctx **ctx*, int *priority*, const char **file*, int *line*, const char **fn*, const char **format*, va_list *args*)

Logging function template

Implement a function with this signature and pass it to set_log_fn() if you want to implement custom logging.

*osd_result* **osd_log_new** (struct osd_log_ctx ** *ctx*, int *log_priority*, *osd_log_fn* *log_fn*)

Create a new instance of osd_log_ctx

You may use the resulting log context on multiple threads.

**See** *osd_log_set_priority()*

**See** *osd_log_set_fn()*

**Parameters**

- `ctx`: the logging context
- `log_priority`: filter: only log messages greater or equal the given priority. Use on the LOG_* constants in stdlog.h Set to 0 to use the default logging priority.
- `log_fn`: logging callback. Set to NULL to disable logging output.

void **osd_log_free** (struct osd_log_ctx ** *ctx_p*)

Free and NULL an osd_log_ctx object

**Parameters**

- `ctx_p`: the log context

void **osd_log_set_fn** (struct osd_log_ctx * *ctx*, *osd_log_fn* *log_fn*)

Set logging function

The built-in logging writes to STDERR. It can be overridden by a custom function to log messages into the user's logging functionality.

In many cases you want the log message to be associated with a context or object of your application, i.e. the object that uses OSD. In this case, set the context or `this` pointer with *osd_log_set_caller_ctx()* and retrieve it inside your `log_fn`.

An example in C++ could look like this:

```
static void MyClass::osdLogCallback(struct osd_log_ctx *gctx,
                                    int priority, const char *file,
                                    int line, const char *fn,
                                    const char *format, va_list args)
{
  MyClass *myclassptr = static_cast<MyClass*>(osd_get_caller_ctx(gctx));
  myclassptr->doLogging(format, args);
}

MyClass::MyClass()
{
  // ...
  osd_log_set_caller_ctx(gctx, this);
  osd_log_set_fn(&MyClass::osdLogCallback);
  // ...
}

MyClass::doLogging(const char*   format, va_list args)
{
```

(continues on next page)

(continued from previous page)

```
    printf("this = %p", this);
    vprintf(format, args);
}
```

**Parameters**

- `ctx`: the log context
- `log_fn`: the used logging function

int **`osd_log_get_priority`**(struct osd_log_ctx * *ctx*)

Get the logging priority

The logging priority is the lowest message type that is reported.

**Return** the log priority

**See** *osd_log_set_priority()*

**Parameters**

- `ctx`: the log context

void **`osd_log_set_priority`**(struct osd_log_ctx * *ctx*, int *priority*)

Set the logging priority

The logging priority is the lowest message type that is reported.

Allowed values for `priority` are `LOG_DEBUG`, `LOG_INFO` and `LOG_ERR` as defined in `syslog.h`.

For example setting `priority` will to `LOG_INFO` will result in all error and info messages to be shown, and all debug messages to be discarded.

**See** *osd_log_get_priority()*

**Parameters**

- `ctx`: the log context
- `priority`: new priority value

void **`osd_log_set_caller_ctx`**(struct osd_log_ctx * *ctx*, void * *caller_ctx*)

Set a caller context pointer

This library does not use this pointer in any way, you're free to set it to whatever your application needs.

**See** *osd_log_get_caller_ctx()*

**See** *osd_log_set_fn()* for a code example using this functionality

**Parameters**

- `ctx`: the log context
- `caller_ctx`: the caller context pointer

void* **`osd_log_get_caller_ctx`**(struct osd_log_ctx * *ctx*)

Get the caller context pointer

**Return** the caller context pointer

**See** *osd_log_set_caller_ctx()*

**Parameters**

- `ctx`: the log context

### 2.5.10 osd_packet class

Representation of a single packet in the debug interconnect.

Accessor functions encapsulate reading and writing fields inside the packet.

#### Usage

```
#include <osd/osd.h>
#include <osd/packet.h>
```

#### Public Interface

libosd-packet::**osd_packet_type**

Packet types

*Values:*

**0**

**1**

**2**

**3**

libosd-packet::**osd_packet_type_reg_subtype**

Values of the TYPE_SUB field in if TYPE == OSD_PACKET_TYPE_REG

*Values:*

**0b0000**

**0b0001**

**0b0010**

**0b0011**

**0b1000**

**0b1001**

**0b1010**

**0b1011**

**0b1100**

**0b0100**

**0b0101**

**0b0110**

**0b0111**

**0b1110**

**0b1111**

libosd-packet::**osd_packet_type_event_subtype**

Values of the TYPE_SUB field in if TYPE == OSD_PACKET_TYPE_EVENT

*Values:*

**0**

**1**

**5**

*osd_result* **osd_packet_new**(struct *osd_packet* ** *packet*, size_t *size_data_words*)

Allocate memory for a packet with given data size and zero all data fields

The osd_packet.size field is set to the allocated size.

**Return** OSD_OK if successful, any other value indicates an error

**See** *osd_packet_new_from_zframe()*

**See** *osd_packet_realloc()*

**See** *osd_packet_free()*

**Parameters**

- `[out] packet`: the packet to be allocated
- `[in] size_data_words`: number of uint16_t words in the packet, including the header words.

*osd_result* **osd_packet_realloc**(struct *osd_packet* ** *packet_p*, size_t *data_size_words_new*)

Reallocate memory for a packet to increase or decrease its size

**Return** OSD_OK if successful, any other value indicates an error

**See** *osd_packet_new()*

**See** *osd_packet_free()*

**Parameters**

- `packet_p`: A pointer to an existing packet, which will be reallocated. The resulting pointer will potentially differ from the the passed pointer!
- `[in] size_data_words`: number of uint16_t words in the packet, including the header words. May be larger or smaller than the size of the existing packet.

*osd_result* **osd_packet_new_from_zframe**(struct *osd_packet* ** *packet*, const zframe_t * *frame*)

Create a new packet from a zframe

**See** *osd_packet_new()*

void **osd_packet_free**(struct *osd_packet* ** *packet*)

Free the memory associated with the packet and NULL the object

*osd_result* **osd_packet_combine**(struct *osd_packet* ** *first_p*, const struct *osd_packet* * *second*)

Append the payload of the second packet into the first packet

The header data of the second packet is ignored.

**Return** OSD_OK if successful, any other value indicates an error

**Parameters**

- `first_p`: a pointer to an existing packet, which will be reallocated the resulting pointer will be *different* than the the passed pointer!
- `second`: the packet of which the payload is appended to `first`

unsigned int **osd_packet_get_dest**(const struct *osd_packet* * *packet*)

Extract the DEST field out of a packet

unsigned int **osd_packet_get_src**(const struct *osd_packet* * *packet*)

Extract the SRC field out of a packet

unsigned int **osd_packet_get_type** (const struct *osd_packet* * *packet*)
    Extract the TYPE field out of a packet

unsigned int **osd_packet_get_type_sub** (const struct *osd_packet* * *packet*)
    Extract the TYPE_SUB field out of a packet

*osd_result* **osd_packet_set_type_sub** (struct *osd_packet* * *packet*, const unsigned int *type_sub*)
    Set the TYPE_SUB field in a packet

*osd_result* **osd_packet_set_header** (struct *osd_packet* * *packet*, const unsigned int *dest*, const unsigned int *src*, const enum osd_packet_type *type*, const unsigned int *type_sub*)
    Populate the header of a *osd_packet*

**Return** OSD_OK on success, any other value indicates an error

**Parameters**

- `packet`:
- `dest`: packet destination
- `src`: packet source
- `type`: packet type
- `type_sub`: packet subtype

size_t **osd_packet_sizeof** (const struct *osd_packet* * *packet*)
    Size in bytes of a packet

unsigned int **osd_packet_sizeconv_payload2data** (unsigned int *payload_words*)
    Get the number of data words required for the given payload words

Data words are the number of 16 bit words in a DI packet, including the DI header. Payload words are the number of 16 bit words in a DI packet excluding the header.

unsigned int **osd_packet_sizeconv_data2payload** (unsigned int *data_words*)
    Get the number of paylaod words required for the given data words

Data words are the number of 16 bit words in a DI packet, including the DI header. Payload words are the number of 16 bit words in a DI packet excluding the header.

void **osd_packet_log** (const struct *osd_packet* * *packet*, struct osd_log_ctx * *log_ctx*, const char * *msg*)
    Log a debug message with the packet in human-readable form

Use the `msg` parameter to prefix the dumped packet in the log entry with, for example, the type of packet being logged. This is preferrable over writing two log entries to keep the information together.

**Parameters**

- `packet`: packet to log
- `log_ctx`: the log context to write to
- `msg`: message to be prepended to the dumped packet

void **osd_packet_dump** (const struct *osd_packet* * *packet*, FILE * *fd*)
    Dump a packet in human-readable (debugging) form to a file stream

**See** *osd_packet_to_string()*

**Parameters**

- `packet`: packet to dump
- `fd`: stream to dump packet to. You can use stdout and stderr here.

void **osd_packet_to_string**(const struct *osd_packet* * *packet*, char ** *str*)

Dump the packet to a string (for human consumption)

The string representation of a packet is for debugging purposes only and may change at any time, do not rely on it for automated parsing.

**See** *osd_packet_dump()*

bool **osd_packet_fwrite**(const struct *osd_packet* * *packet*, FILE * *fd*)

Write a packet from a file descriptor

In its current implementation this function memory-dumps *osd_packet* structs to a file, without any encoding. This has some consequences:

- The resulting file is in native endianness, i.e. not portable between little and big endian machines.
- No file header is present, or any for of file magic number to identify the file type.
- No file integrity checks, such as checksums.

**Return** bool operation successful?

**See** *osd_packet_fread()*

**Parameters**

- `packet`: the packet to write
- `fd`: the open file descriptor to write to

struct *osd_packet** **osd_packet_fread**(FILE * *fd*)

Read a packet from an open file descriptor

See the discussion in *osd_packet_fwrite()* for known limitations.

**Return** the read packet, or NULL if reading failed

**See** *osd_packet_fwrite()*

**Parameters**

- `fd`: an open file descriptor to read from

bool **osd_packet_equal**(const struct *osd_packet* * *p1*, const struct *osd_packet* * *p2*)

Check if two packets are equal

**DP_HEADER_TYPE_SHIFT**

**DP_HEADER_TYPE_MASK**

**DP_HEADER_TYPE_SUB_SHIFT**

**DP_HEADER_TYPE_SUB_MASK**

**DP_HEADER_SRC_SHIFT**

**DP_HEADER_SRC_MASK**

**DP_HEADER_DEST_SHIFT**

**DP_HEADER_DEST_MASK**

**struct osd_packet**

*#include <packet.h>* A packet in the Open SoC Debug system

### 2.5.11 Error Handling

All functions in this project use a common scheme for return codes and error handling.

All functions, except for getters and setters, return the type `osd_result`. A return code of `OSD_OK` indicates a successful function call, any other return value indicates an error. The error types are given as one of the *OSD_ERROR_**.

Use `OSD_SUCCEEDED` to check if a function call was successful, and `OSD_FAILED` to check if it failed.

#### Use of assertions

Using assertions within libosd causes the application using libosd to crash. A crash can be a good way to fail early, however, it may be used only for non-recoverable errors or programming errors. In consequence, libosd makes use of assertions to check for internal errors (i.e. bugs in libosd), or to check for violated API calling conventions. However, assertions are not used to check for invalid or missing input data from known-to-be-unreliable sources (e.g. from the network, from a user or from the file system).

Assertions are used in the following cases:

- API calling conventions are not followed.
- Violoated pre- or postconditions: the internal state when entering or exiting a function isn't what was expected.

Assertions are *not* used in in the following scenarios:

- User input to a function is not valid.
- Unexpected data was received from the network, from the file system, or similar sources.
- The communication broke down.

#### Handling memory allocation errors

In general, out of memory errors returned from *malloc()* and similar functions are fatal, i.e. cause the program using libosd to crash. This is motivated by the fact that these errors occur very rarely and the associated error handling path is rarely executed and tested.

In some cases where larger memory allocations are required, especially contiguous chunks of memory, error handling is implemented and fallback paths are used.

#### Example

```c
#include <osd/osd.h>
#include <assert.h>

osd_result rv;

// common case: pass error on to calling function
rv = osd_some_call();
if (OSD_FAILED(rv)) {
  return rv;
}

// check for a successful call
rv = osd_another_call();
if (OSD_SUCCEEDED(rv)) {
  printf("Call successful!\n");
}
```

(continues on next page)

(continued from previous page)

```
// A failing call may not happen and hence is a bug in OSD: use assert()
rv = osd_must_succeed_call();
assert(OSD_SUCCEEDED(rv));

// fatal malloc: assert if the memory allocation failed
char* some_data = malloc(sizeof(char) * 100));
assert(some_data);
```

### Public Interface

**typedef int osd_result**
Standard return type

**OSD_OK**
Return code: The operation was successful

**OSD_ERROR_FAILURE**
Return code: Generic (unknown) failure

**OSD_ERROR_DEVICE_ERROR**
Return code: debug system returned a failure

**OSD_ERROR_DEVICE_INVALID_DATA**
Return code: received invalid or malformed data from device

**OSD_ERROR_COM**
Return code: failed to communicate with device

**OSD_ERROR_TIMEDOUT**
Return code: operation timed out

**OSD_ERROR_NOT_CONNECTED**
Return code: not connected to the device

**OSD_ERROR_PARTIAL_RESULT**
Return code: this is a partial result, not all requested data was obtained

**OSD_ERROR_ABORTED**
Return code: operation aborted

**OSD_ERROR_CONNECTION_FAILED**
Return code: connection failed

**OSD_ERROR_OOM**
Return code: Out of memory

**OSD_ERROR_FILE**
Return code: file operation failed

**OSD_ERROR_MEM_VERIFY_FAILED**
Return code: memory verification failed

**OSD_ERROR_WRONG_MODULE**
Return code: unexpected module type

**OSD_FAILED**(rv)
Return true if |rv| is an error code

**OSD_SUCCEEDED**(rv)
Return true if |rv| is a successful return code

### 2.5.12 osd_memaccess class

Access a memory in the system (high-level API).

#### Usage

```
#include <osd/osd.h>
#include <osd/memaccess.h>
```

#### Public Interface

#### Functions

*osd_result* **osd_memaccess_new** (struct osd_memaccess_ctx ** *ctx*, struct osd_log_ctx * *log_ctx*, const char * *host_controller_address*)

Create a new context object

*osd_result* **osd_memaccess_connect** (struct osd_memaccess_ctx * *ctx*)

Connect to the host controller

**Return** OSD_OK on success, any other value indicates an error

**See** *osd_hostmod_disconnect()*

**Parameters**

- `ctx`: the osd_hostmod_ctx context object

*osd_result* **osd_memaccess_disconnect** (struct osd_memaccess_ctx * *ctx*)

Shut down all communication with the device

**Return** OSD_OK on success, any other value indicates an error

**See** osd_hostmod_run()

**Parameters**

- `ctx`: the osd_hostmod context object

bool **osd_memaccess_is_connected** (struct osd_memaccess_ctx * *ctx*)

Is the connection to the device active?

**Return** 1 if connected, 0 if not connected

**See** *osd_hostmod_connect()*

**See** *osd_hostmod_disconnect()*

**Parameters**

- `ctx`: the osd_hostmod context object

void **osd_memaccess_free** (struct osd_memaccess_ctx ** *ctx_p*)

Free the context object

*osd_result* **osd_memaccess_cpus_start** (struct osd_memaccess_ctx * *ctx*, unsigned int *subnet_addr*)

(Re-)Start all CPUs in the subnet

**Return** OSD_OK on success, any other value indicates an error

**See** *osd_memaccess_cpus_stop()*

**Parameters**

- `ctx`: the context object
- `subnet_addr`: the subnet to start all CPUs in

*osd_result* **`osd_memaccess_cpus_stop`** (struct osd_memaccess_ctx * *ctx*, unsigned int *subnet_addr*)

Stop all CPUs in the given subnet to avoid interfering with the memory access

**Return** OSD_OK on success, any other value indicates an error

**See** *osd_memaccess_cpus_start()*

**Parameters**

- `ctx`: the context object
- `subnet_addr`: the subnet to stop all CPUs in

*osd_result* **`osd_memaccess_find_memories`** (struct osd_memaccess_ctx * *ctx*, unsigned int *subnet_addr*, struct *osd_mem_desc* ** *memories*, size_t * *num_memories*)

Get all memories in a subnet

**Return** OSD_OK if the operation was successful OSD_ERROR_PARTIAL_RESULT not all memories could be enumerated any other value indicates an erorr

**Parameters**

- `ctx`: the context object
- `subnet_addr`: the subnet to find memories in
- `[out] the`: memories found in the subnet
- `[out] the`: number of entries in the `memories` array

*osd_result* **`osd_memaccess_loadelf`** (struct osd_memaccess_ctx * *ctx*, const struct *osd_mem_desc* * *mem_desc*, const char * *elf_file_path*, bool *verify*)

Load an ELF file into a memory

**Parameters**

- `ctx`: the context object
- `mem_desc`: the memory to load the data into
- `elf_file_path`: file system path to the ELF file to be loaded
- `verify`: verify the write operation by reading the file back and and compare the data.
- `OSD_OK`: if successful, any other value indicates an error

### 2.5.13 osd_systracelogger class

Obtain and decode a system trace (high-level API).

System traces are typically obtained by a STM module, which in turn gathers this information using processor-specific mechanisms (such as observing nop instructions, or writes to special registers). System traces are streams of (id, value) tuples. Some ids have a specification-defined meaning and can be decoded in a special way. Most commonly, id 4 is used to transmit a single printed character from the software, providing an easy way to "print" data from software through the debug system to the host PC. *osd_systracelogger* contains functions to decode such streams of characters and write them to a file.

### Usage

```
#include <osd/osd.h>
#include <osd/systracelogger.h>
```

### Public Interface

### Functions

*osd_result* **osd_systracelogger_new** (struct osd_systracelogger_ctx ** *ctx*, struct osd_log_ctx * *log_ctx*, const char * *host_controller_address*, uint16_t *stm_di_addr*)
Create a new context object

*osd_result* **osd_systracelogger_connect** (struct osd_systracelogger_ctx * *ctx*)
Connect to the host controller

**Return** OSD_OK on success, any other value indicates an error

**See** *osd_hostmod_disconnect()*

**Parameters**

- `ctx`: the osd_hostmod_ctx context object

*osd_result* **osd_systracelogger_disconnect** (struct osd_systracelogger_ctx * *ctx*)
Shut down all communication with the device

**Return** OSD_OK on success, any other value indicates an error

**See** osd_hostmod_run()

**Parameters**

- `ctx`: the osd_hostmod context object

bool **osd_systracelogger_is_connected** (struct osd_systracelogger_ctx * *ctx*)
Is the connection to the device active?

**Return** 1 if connected, 0 if not connected

**See** *osd_hostmod_connect()*

**See** *osd_hostmod_disconnect()*

**Parameters**

- `ctx`: the osd_hostmod context object

void **osd_systracelogger_free** (struct osd_systracelogger_ctx ** *ctx_p*)
Free the context object

*osd_result* **osd_systracelogger_start** (struct osd_systracelogger_ctx * *ctx*)
Start collecting system logs

Instruct the STM module to start sending traces to us.

*osd_result* **osd_systracelogger_stop** (struct osd_systracelogger_ctx * *ctx*)
Stop collecting system logs

*osd_result* **osd_systracelogger_set_sysprint_log** (struct osd_systracelogger_ctx * *ctx*, FILE * *fp*)
Set a file to write all sysprint output to

*osd_result* **osd_systracelogger_set_event_log**(struct osd_systracelogger_ctx * *ctx*, FILE * *fp*)
Set a file to write all received STM events to

### 2.5.14 osd_coretracelogger class

Obtain and decode a core trace (high-level API).

#### Usage

```
#include <osd/osd.h>
#include <osd/coretracelogger.h>
```

#### Public Interface

#### Functions

*osd_result* **osd_coretracelogger_new**(struct osd_coretracelogger_ctx ** *ctx*, struct osd_log_ctx * *log_ctx*, const char * *host_controller_address*, uint16_t *ctm_di_addr*)
Create a new context object

*osd_result* **osd_coretracelogger_connect**(struct osd_coretracelogger_ctx * *ctx*)
Connect to the host controller

**Return** OSD_OK on success, any other value indicates an error

**See** *osd_hostmod_disconnect()*

**Parameters**

- `ctx`: the osd_hostmod_ctx context object

*osd_result* **osd_coretracelogger_disconnect**(struct osd_coretracelogger_ctx * *ctx*)
Shut down all communication with the device

**Return** OSD_OK on success, any other value indicates an error

**See** osd_hostmod_run()

**Parameters**

- `ctx`: the osd_hostmod context object

bool **osd_coretracelogger_is_connected**(struct osd_coretracelogger_ctx * *ctx*)
Is the connection to the device active?

**Return** 1 if connected, 0 if not connected

**See** *osd_hostmod_connect()*

**See** *osd_hostmod_disconnect()*

**Parameters**

- `ctx`: the osd_hostmod context object

void **osd_coretracelogger_free**(struct osd_coretracelogger_ctx ** *ctx_p*)
Free the context object

*osd_result* **`osd_coretracelogger_start`** (struct osd_coretracelogger_ctx * *ctx*)

Start collecting system logs

Instruct the STM module to start sending traces to us.

*osd_result* **`osd_coretracelogger_stop`** (struct osd_coretracelogger_ctx * *ctx*)

Stop collecting system logs

*osd_result* **`osd_coretracelogger_set_log`** (struct osd_coretracelogger_ctx * *ctx*, FILE * *fp*)

Set a file to write all log output to

**Return** OSD_OK if successful, any other value indicates an error

**Parameters**

- `ctx`: context object
- `fp`: a file pointer to write the logs to

*osd_result* **`osd_coretracelogger_set_elf`** (struct osd_coretracelogger_ctx * *ctx*, const char * *elf_filename*)

Set the path to the ELF file used to decode the core trace events

To disable ELF parsing, set elf_filename to NULL.

**Return** OSD_OK when reading the ELF file succeeded any other value indicates an error

**Parameters**

- `ctx`: context object
- `elf_filename`: path to the ELF file. Set to NULL to disable ELF parsing.

CHAPTER 3

# OSD Software User Guides

**Todo:** missing content

man-page like documentation for the tools in tools/*

# Index