White Rabbit Trigger Distribution (WRTD) is a generic framework for distributing triggers (Events) over a White Rabbit (WR) network.

As can be seen in Fig. 1.1, WRTD Nodes receive “input” Events and distribute them to other Nodes over WR in the form of network Messages that are used to transfer the Timestamp of the input Event. The receiving Nodes are programmed to execute some “output” Event (action) upon reception of a particular Message, potentially with some fixed delay added to the Timestamp.

There are two main categories of WRTD applications:

1. A “source” Node receives an input Event, adds a fixed delay to its Timestamp and distributes it to other Nodes. As long as the fixed delay added is greater than the upper-bound latency of the network (a fundamental feature of WR itself), all receiving nodes will receive the Message before the programmed time and will execute simultaneously their action (thanks to the sub-ns synchronisation provided by WR).

2. The receiving Nodes are “recording” devices (e.g. digitisers), capable of storing data in a recording buffer. The source Node transmits the Message, with or without a fixed delay. When one of the destination Nodes receives the Message, it stops recording and rolls-back its buffer to the moment specified by the Timestamp in the received Message (provided that it has a large enough buffer to compensate for the latency). Thus, all Nodes will deliver recorded data from the moment in the past when the input Event was originally received at the source Node.

Of course, the above list is not exhaustive, there are many other potential applications but they are usually permutations of one of the above scenarios.

In WRTD, the programming of Events, Messages and associated actions is done by defining Rules. A Rule simply declares a relationship between an input (cause) and an output (effect) Event. A Rule can state that when a specific Event is received a Message should be transmitted or, that when a Message is received an output Event should be generated. This is depicted in Fig. 1.2.

Section 3 provides a more elaborate discussion on the various basic concepts of WRTD.
Fig. 1.1: Overview of WRTD
Fig. 1.2: Inside a WRTD Node
1.1 Relation to IVI and LXI

LAN eXtensions for Instrumentation (LXI) is a standard, defining a communication protocol for the remote control of instrumentation over an Ethernet-based LAN.

Version 1.5 of the LXI standard splits the specification in two parts:

1. The LXI Core Specification, to which all LXI Devices must conform.
2. A set of optional Extended Functions, to which vendors may choose to conform.

Among these “Extended Functions”, several are related to the synchronisation (via IEEE-1588 PTP) and exchange of real-time event messages between instruments. These include:

- LXI Event Log Extended Function
- LXI Timestamped Data Extended Function
- LXI Clock Synchronization Extended Function
- LXI Event Messaging Extended Function

The core specification requires (Rule 6.1) that all LXI devices provide an Interchangeable Virtual Instruments (IVI) driver. Furthermore, it requires (Rule 6.1.1) that all LXI devices supporting the exchange of event messages, do so by providing an API that conforms to the IVI-3.15 IviLxiSync Specification.

Since the LXI event exchanging mechanism is conceptually very close to WRTD, it was decided to design WRTD to be as close to LXI as possible. In particular:

- WRTD uses the same Message format. This already allows LXI and WRTD devices on the same network to exchange events, even if the API for programming these events is different.
- the WRTD library API mimics that of an IVI driver, with a strong influence from the IVI-3.15 IviLxiSync Specification, even if several of the Repeated Capabilities and Attributes are different.

Hint: Do not worry if you do not understand some of the terminology yet. It will be explained in Section 3.

In the future, and with White Rabbit being standardised within the next release of IEEE-1588, it is foreseen to try to merge WRTD with IVI/LXI. A possible way to do this would be to add a new IVI specification, similar to IVI-3.15, describing the API to control WRTD-enabled devices. This API would be an extension, allowing any instrument with an IVI driver and a WR interface to exchange event messages with any other WRTD node.

1.2 Document Structure

The following is a description of how the remainder of this document is structured (in reverse order).

- Section 6 provides guidelines on how to develop a WRTD Node, including hardware (Section 6.2), gateware (Section 6.3) and firmware (Section 6.4).
• Before you embark however on a new design, please have a look first at the existing reference designs; it could be that one of them is appropriate for your task. Section 5 presents the currently available reference WRTD Nodes that come pre-programmed with their gateware and firmware.

• Whether you develop your own Node or use one of the reference Nodes, Section 4 describes how to access and control your Node. Section 4.1 provides all the details on how to use the C library to develop your own applications. Alternatively, Section 4.2 presents a Python wrapper to the C library that can be used to develop Python-based applications. Section 4.3 describes the generic tools that are built using the Python wrapper and that provide access to a WRTD node without the need to develop any application.

• Section 3 introduces the various basic concepts of WRTD. These concepts are used throughout this document and are fundamental to understanding how WRTD works (and, by extension, how to use it).

• Last but not least, Section 2 takes you through the necessary steps to setup the hardware and install the necessary software.

1.3 Documentation License

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/4.0/.
CHAPTER 2

Installation

Note: Because at the time of the release of WRTD v1.0, the OHWR deployment procedures were undergoing significant changes (in particular with respect to packaging), up-to-date installation instructions will be available through the project Wiki. Once these procedures have been finalised and tested, the contents of the wiki page will be merged here.

2.1 Permissions

By default, WRTD installs with read/write permissions for the root account only.

For users to be able to access the Nodes with a non-root account, it is necessary to set appropriately the file permissions of all devices under /dev/moxturtle.

Note that this is handled automatically when WRTD is installed through packages.
This section introduces the various basic concepts of WRTD. These concepts are used throughout this document and are fundamental to understanding how WRTD works (and, by extension, how to use it).

### 3.1 Node

WRTD is made of Nodes, connected to each other over a WR network. Nodes receive input Events and send output Events. See also Fig. 1.1.

Every Node has Local Channel inputs and/or outputs allowing it to interact with its environment. It also has a connection to a WR network, allowing it to send and/or receive Messages to other Nodes.

### 3.2 Event

Events represent inputs and outputs of a WRTD Node. Input Events received on a Node will result in an output Event to be generated (assuming that the relevant Rule exists to associate an input to an output). In that sense, inputs Events are the causes, while output Events are the effects.

Input Events can be either a Local Channel, an inbound network Message, or an Alarm.

Output Events can be either a Local Channel, or an outbound network Message.

An Event is essentially a combination of an Event ID (the “what”) and an Event Timestamp (the “when”).

#### 3.2.1 Event ID

Every Event is represented by an ID.
Event IDs are 15-character, null-terminated (for a total length of 16) strings that uniquely identify an input or output Event.

*Local Channel* inputs always use an Event ID in the form of \texttt{LC-I<x>}, where \(x\) is a number starting from 1 (e.g. \texttt{LC-I1}).

*Local Channel* outputs always use an Event ID in the form of \texttt{LC-O<x>}, where \(x\) is a number starting from 1 (e.g. \texttt{LC-O1}).

An *Alarm* Event ID will always have a prefix of \texttt{alarm} or \texttt{ALARM}, followed by any other characters (always limited to 16 characters, including null-termination).

All other Event IDs are considered to refer to network *messages*.

**Attention:** An Event ID that is longer than 16 characters (including null-termination) or that is filled with null characters is invalid.

**Note:** Rule 6.4.4 of the *LXI Core Specification* also defines some reserved Event IDs. These are:

- \texttt{LXI<x>} with \(x\) being an integer between 0 and 7 (e.g. \texttt{LXI5})
- \texttt{LAN<x>} with \(x\) being an integer between 0 and 7 (e.g. \texttt{LAN3})
- \texttt{LXIERROR}

Users are advised to not use these identifiers in their own applications.

### 3.2.2 Event Timestamp

Every *Event* has an associated Timestamp.

For an input *Event*, the Timestamp typically represents the moment in time when the *Event* happened.

For an output *Event*, the Timestamp typically represents the moment in time when the *Event* should happen.

Timestamps are expressed using a 48-bit counter for seconds, a 32-bit counter for nanoseconds (which is reset every time the counter reaches \(10^9\)) and a 16-bit counter for fractional nanoseconds (where every “tick” represents \(2^{-16}\)ns).

**Hint:** All Timestamps represent TAI time since 00:00:00 Thursday, 1 January 1970 (Unix Epoch time).

**Hint:** For most applications, the upper 16 bits of the seconds counter can be ignored/assumed to be zero. A 32-bit seconds counter that was started on 00:00:00 Thursday, 1 January 1970 will overflow on 06:26:16 Sunday, 7 February 2106.
### 3.3 Event Log

The Event Log records information about all received and transmitted *Events*, as well as information regarding any discarded *Event*, along with the reason for discarding it.

The Event Log has a limited storage buffer. Newer entries will overwrite older, unread ones.

For an explanation of the fields of an Event Log entry, please refer to the documentation of the `wrtd_get_next_event_log_entry()` function.

### 3.4 Local Channel

Local Channels represent the connections of a *Node* to its environment. They can be either inputs or outputs.

A Local Channel input delivers input Events to the *Node*. Typical examples include the external trigger input of a digitiser, a Time to Digital Converter (TDC) or a TTL input channel on a digital I/O board.

A Local Channel output transmits output Events from the *Node*. Typical examples include a Fine Delay generator or a TTL output channel on a digital I/O board.

All Local Channels use specific IDs as described in *Event IDs*.

### 3.5 Message

WRTD Event Messages (or, simply, Messages) follow the LXI Event Messaging format, as defined in Rule 4.3 of the LXI Event Messaging Extended Function specification.

To ensure compatibility and interoperability with LXI devices, WRTD Event Messages are transmitted using multicast UDP on address **224.0.23.159**, port **5044** (Rule 3.3.1 of the specification).

Each Message is transmitted as a single Ethernet frame, with a UDP header and a payload as shown in [Fig. 3.1](#).

| Octet | Octet | Bit   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| 0     | 0     | 0x80 ('I') | 0x58 ('X') | 0x49 ('I') | 0x00 |
| 8     | 64    |                      |                      |                      |                      |
| 12    | 96    |                      |                      |                      |                      |
| 16    | 128   |                      |                      |                      |                      |
| 20    | 160   | Sequence number |                      |                      |                      |
| 24    | 192   | Event timestamp seconds (32 lower bits) |                      |                      |                      |
| 28    | 224   | Event timestamp nanoseconds |                      |                      |                      |
| 32    | 256   | Event timestamp fractional nanoseconds | Event timestamp seconds (16 upper bits) |                      |                      |
| 36    | 288   | 0x00 | 0x00 | 0x00 | 0x00 | 0x00

**Fig. 3.1**: Contents of a WRTD Event Message

The contents of a WRTD Event Message are again based on LXI Event Messages, with the “Domain” and “Flag” fields (octets 3 and 36 respectively) fixed to zero.
Each Message contains an Event ID, an Event Timestamp and a sequence number. The latter is a counter that gets increased by one every time a new Event is generated.

**Hint:** The sequence counter can be used to detect lost or duplicate Messages.

**Hint:** Although there are currently no “Data Fields” defined (octets 37 and beyond), it should be highlighted that the LXI Event message format supports an arbitrary number of data fields, in the form of Type/Length/Value (TLV) triplets, which could be used to provide additional functionality to WRTD in the future.

Please refer to Section 4.3 of the LXI Event Messaging Extended Function specification for more details.

### 3.6 Repeated Capability

IVI uses the term Repeated Capability to express functionality that is duplicated in an instrument, with each instance potentially having different settings. A typical example of a Repeated Capability is a channel of an oscilloscope; the instrument has many channels, each one offering the exact same functionality, but each one with its own settings (Attributes).

Furthermore, IVI defines the API for accessing Repeated Capabilities in Section 12 of IVI-3.4: API Style Guide.

WRTD defines the following Repeated Capabilities for each Node:

- Rule (see also Rules API)
- Alarm (see also Alarms API)
- Application (see also Applications API)

### 3.6.1 Repeated Capability ID

An instance of a Repeated Capability is designated by a unique Repeated Capability ID.

WRTD uses the Parameter Style API, defined in Section 12.1 of IVI-3.4: API Style Guide, where each function related to a Repeated Capability expects the ID of the Repeated Capability instance as a parameter. This parameter is also called a Repeated Capability Selector in IVI terminology.

**Note:** Similar to an Event ID, a Repeated Capability ID is limited to 16 characters, including null termination.

**Hint:** Section 4.4 of IVI-3.1: Driver Architecture Specification shows that a Selector can include groups of IDs, ranges, nested IDs and much more. For now, WRTD only supports simple selectors, allowing a single ID to be selected at a time, but this could change in future releases.


3.7 Attribute

WRTD uses Attributes to represent the various settings of the Node and defines get/set functions to access them. This behaviour is identical to IVI.

Attributes can be of one of the following types:

- Boolean
- 32-bit Integer
- String
- Timestamp (new type, does not exist in IVI)

Attributes can be attached to a Repeated Capability, or they can be “global” (apply to the whole Node).

**Note:** Since global Attributes are not attached to any Repeated Capability, when using one of the functions to get/set a global Attribute, a special Repeated Capability ID (WRTD_GLOBAL_REP_CAP_ID) must be passed to the function as the Selector.

Please refer to the Attribute Handling API for more details.

3.8 Rule

A Rule is a Repeated Capability instance inside a Node that links input to output Events.

Rules are declared (and deleted) using the Rules API. Their configuration is controlled via Attributes that can be manipulated using the Attribute Handling API.

When an input Event is received by a Node, WRTD tries to match it with any declared (and enabled) Rule. The process that is followed for each input Event is depicted in Fig. 3.2.

![Fig. 3.2: Rule processing for incoming Events. The red hexagons represent Attributes.](image)

Once an input Event has been matched and all delays have been applied to it, it is forwarded to the next processing block that generates the preconfigured output Event. This is depicted in Fig. 3.3.
Fig. 3.3: Rule processing for outgoing *Events*. The red hexagons represent *Attributes*.

**Hint:** There are actually more *Attributes* than the ones shown in Fig. 3.2 and Fig. 3.3. Please refer to the *Attribute Handling API* for the complete list, as well as an explanation of each *Attribute*.

### 3.9 Alarm

An *Alarm* is simply a user-defined moment to generate internally an input *Event*. Similar to a *Rule*, an Alarm is a *Repeated Capability* instance inside a *Node*.

Alarms are declared (and deleted) using the *Alarms API*. Their configuration is controlled via *Attributes* that can be manipulated using the *Attribute Handling API*.

**Note:** The *Repeated Capability ID* of an Alarm must always have a prefix of *alarm* or *ALARM*, followed by any other characters (always limited to 16 characters, including null-termination).

The *Event ID* of the input *Event* generated by an Alarm will match its *Repeated Capability ID*.

Every *Node* checks periodically if any of the declared (and enabled) Alarms need to be triggered, following the process depicted in Fig. 3.4.

**Hint:** There are actually more *Attributes* than the ones shown in Fig. 3.4. Please refer to the *Attribute Handling API* for the complete list, as well as an explanation of each *Attribute*.

### 3.10 Application

A *Node* may be running one or more Applications (firmware). As an example, the *SPEC150T-based FMC-ADC* runs a single Application (to communicate with the ADC), while the *SVEC-based TDC+FD* runs two Applications (one for the TDC and one for the Fine Delay generator).
As such, an Application is also a *Repeated Capability* instance inside a *Node*.

Unlike *Rules* and *Alarms*, users cannot declare or remove Applications. They can only get their (read-only) *Attributes* which provide information regarding firmware version, number and direction of *Local Channels*, etc.

Please refer to the *Attribute Handling API* for the complete list of Application-related *Attributes*. The *Repeated Capability ID* of each Application can be retrieved using the *Applications API*. 

![Diagram of Alarm Processing](image-url)
There are three options offered for accessing a WRTD Node (in decreasing order of flexibility and complexity):

- By writing your own application using the provided C Library.
- By writing your own application using the provided Python Wrapper.
- By using the provided Tools.

### 4.1 C Library

The WRTD C Library is the standard, most flexible way of accessing a WRTD Node. The library itself is modelled after IVI and LXI (see also Relation to IVI and LXI).

The following specifications are relevant to and used by WRTD:

- IVI-3.1 Driver Architecture Specification
- IVI-3.2 Inherent Capabilities Specification
- IVI-3.4 API Style Guide
- IVI-3.15 IviLxiSync Specification
- LXI Core Specification
- LXI Event Messaging Extended Function

#### 4.1.1 Error Handling API

Every function in the WRTD C library returns an error code of type `wrtd_status`. The Error Handling API provides functions for retrieving and interpreting errors from the Node. Error codes can be converted to strings by means of the `wrtd_error_message()` function. The latest error can be retrieved with the `wrtd_get_error()` function.
Error Codes

enum wrtd_status

White Rabbit Trigger Distribution error codes.

Names and values inspired by IVI-3.2 and IVI-3.15.

Values:

WRTD_SUCCESS = 0
No error.

__WRTD_ERROR_BASE = 0xBFFA0000
Same as IVI_INHERENT_ERROR_BASE.

WRTD_ERROR_INVALID_ATTRIBUTE = __WRTD_ERROR_BASE + 0x0C
Invalid/unknown attribute.

WRTD_ERROR_ATTR_NOT_WRITEABLE = __WRTD_ERROR_BASE + 0x0D
Attempt to write a read-only attribute.

WRTD_ERROR_ATTR_NOT_READABLE = __WRTD_ERROR_BASE + 0x0E
Attempt to read a write-only attribute.

WRTD_ERROR_INVALID_VALUE = __WRTD_ERROR_BASE + 0x10
Invalid value provided.

WRTD_ERROR_NOT_INITIALIZED = __WRTD_ERROR_BASE + 0x1D
Device not initialized.

WRTD_ERROR_UNKNOWN_CHANNEL_NAME = __WRTD_ERROR_BASE + 0x20
Unknown channel name.

WRTD_ERROR_OUT_OF_MEMORY = __WRTD_ERROR_BASE + 0x56
Device out of memory.

WRTD_ERROR_NULL_POINTER = __WRTD_ERROR_BASE + 0x58
Null pointer.

WRTD_ERROR_UNEXPECTED_RESPONSE = __WRTD_ERROR_BASE + 0x59
Unexpected response.

WRTD_ERROR_Resource_UNKNOWN = __WRTD_ERROR_BASE + 0x60
Unknown resource.

WRTD_ERROR_BADLY_FORMED_SELECTOR = __WRTD_ERROR_BASE + 0x66
Incorrect repeated capability selector.

__WRTD_LXISYNC_ERROR_BASE = 0xBFFA3000
Same as IVI_LXISYNC_ERROR_BASE.

WRTD_ERROR_ALARM_EXISTS = __WRTD_LXISYNC_ERROR_BASE + 0x07
The alarm already exists.

WRTD_ERROR_ALARM_Does_NOT_EXIST = __WRTD_LXISYNC_ERROR_BASE + 0x08
The specified alarm has not been defined.

__WRTD_SPECIFIC_ERROR_BASE = 0xBFFA6000
Same as IVI_VENDOR_SPECIFIC_ERROR_BASE.
WRTD_ERROR_VERSION_MISMATCH = __WRTD_SPECIFIC_ERROR_BASE + 0x00
Version mismatch.

WRTD_ERROR_INTERNAL = __WRTD_SPECIFIC_ERROR_BASE + 0x01
Internal error.

WRTD_ERROR_UNKNOWN_LOG_TYPE = __WRTD_SPECIFIC_ERROR_BASE + 0x02
Unknown event log type/reason.

WRTD_ERROR_RESOURCE_ACTIVE = __WRTD_SPECIFIC_ERROR_BASE + 0x03
Resource is not disabled and cannot be changed.

WRTD_ERROR_ATTR_GLOBAL = __WRTD_SPECIFIC_ERROR_BASE + 0x04
Attempt to access a global attribute without using the global attribute selector.

WRTD_ERROR_OUT_OF_RESOURCES = __WRTD_SPECIFIC_ERROR_BASE + 0x05
The device has no more resources to allocate.

WRTD_ERROR_RULE_EXISTS = __WRTD_SPECIFIC_ERROR_BASE + 0x06
The rule already exists.

WRTD_ERROR_RULE_DOES_NOT_EXIST = __WRTD_SPECIFIC_ERROR_BASE + 0x07
The specified rule has not been defined.

__WRTD_ERROR_MAX_NUMBER
Always last entry in this enum.

Functions

wrtd_status wrtd_get_error(wrtd_dev *wrtd, wrtd_status *error_code, int32_t error_description_buffer_size, char *error_description)
Retrieve and clear the last error from the device.
Modelled after the IVI-C GetError function.
This function complies with IVI-3.2, section 3.1.2.1 (Additional Compliance Rules for C Functions with ViChar Array Output Parameters), with the exception of the buffer_size < 0 case, which produces an error instead of allowing a potential buffer overflow.

Return wrtd_status. However, if the buffer size parameter is 0, then this function returns instead a positive value, indicating the minimum buffer size necessary to fit the full message. See also IVI-3.2, section 3.1.2.1.

Parameters

• wrtd: Device token.
• error_code: wrtd_status pointer to return the error code. Ignored if NULL.
• error_description_buffer_size: Size of pre-allocated error_description buffer.
• error_description: Buffer to store the detailed error message string.

wrtd_status wrtd_error_message(wrtd_dev *wrtd, wrtd_status err_code, char *err_message)
Convert a wrtd_status error code to a string.
Modelled after the IVI-C ErrorMessage function.

4.1. C Library
Return \texttt{wrtd\_status}

Parameters

- \texttt{wrtd}: Device token. Can be NULL to allow calling this function even when initialisation has failed.
- \texttt{err\_code}: \texttt{wrtd\_status} error code to convert.
- \texttt{err\_message}: Buffer of at least 256 bytes to store the resulting string.

Listing 4.1: Error handling

```c
#include <libwrtd.h>

int main(void) {
    wrtd_dev *wrtd;
    wrtd_status status, err_code;
    char status_str[256];
    char *err_msg;
    int buf_size;

    status = wrtd_init(1, false, NULL, &wrtd);
    if (status != WRTD_SUCCESS) {
        /* use \texttt{wrtd\_error\_message} because \texttt{wrtd\_init} failed 
            and the \texttt{wrtd} pointer is not valid. */
        wrtd_error_message(wrtd, status, status_str);
        printf("ERROR: %d, %s\n", status, status_str);
        return status;
    }

    status = wrtd_get_attr_bool(wrtd, WRTD_GLOBAL_REP_CAP_ID, 
                                WRTD_ATTR_EVENT_LOG_EMPTY);
    if (status != WRTD_SUCCESS) {
        /* query the necessary buffer size for 
           the error message */
        buf_size = wrtd_get_error(wrtd, NULL, 0, NULL);
        /* allocate the buffer */
        err_msg = calloc(sizeof(char), buf_size);
        /* retrieve the error code and message */
        wrtd_get_error(wrtd, &err_code, buf_size, err_msg)
        printf("ERROR: %d, %s\n", err_code, err_msg);
        return status;
    }

    wrtd_close(wrtd);

    return 0;
}
```

Important: In the remaining code examples we omit error checking on purpose, to simplify the examples. However, in a real application, users should always check the status code of every
call to a WRTD function, like in Listing 4.1.

### 4.1.2 Initialisation API

The initialisation API provides the functions to initiate/close a connection to the Node, as well as to reset it.

`wrtd_init()` is the first function that should be called by any program, in order to obtain the “device token” to be used in all subsequent function calls.

Conversely, `wrtd_close()` should be called before exiting the program. No further WRTD library functions can be used after that.

In order to identify the Node to connect to, it is necessary to provide the ID of that Node. This ID is simply an integer that uniquely identifies a Node within a given host system. Functions `wrtd_get_node_count()` and `wrtd_get_node_id()` can help you figure out the ID of each Node.

**Important:** The Node ID is not sequential, nor does it start counting from zero (or one). It might well be that you only have one Node in a given host, and that it has an ID different than 1. Always retrieve therefore the Node ID with `wrtd_get_node_id()`.

```c
wrtd_status wrtd_init (uint32_t node_id, bool reset, const char *options_str, wrtd_dev **wrtd)
```

Initialize the WRTD Node and obtain the WRTD device token.

**Return** `wrtd_status`

**Parameters**

- node_id: WRTD Node ID
- reset: Reserved for future use.
- options_str: Reserved for future use.
- wrtd: Pointer to WRTD device token.

```c
wrtd_status wrtd_close (wrtd_dev *wrtd)
```

Close a WRTD Node and release all resources.

**Return** `wrtd_status`

**Parameters**

- wrtd: Device token.

```c
wrtd_status wrtd_reset (wrtd_dev *wrtd)
```

Reset a WRTD Node. This will remove all defined Alarms and Rules.

**Return** `wrtd_status`

**Parameters**

- wrtd: Device token.
\texttt{wrtd\_status wrtd\_get\_node\_count (uint32\_t *count)}

Retrieve the number of detected WRTD Nodes.

\textbf{Return} \texttt{wrtd\_status}

\textbf{Parameters}

- \texttt{count}: number of detected WRTD Nodes

\texttt{wrtd\_status wrtd\_get\_node\_id (uint32\_t index, uint32\_t *node\_id)}

Retrieve the ID of a WRTD Node.

Before calling this function, you should probably call \texttt{wrtd\_get\_node\_count} to know the number of Nodes.

\textbf{Return} \texttt{wrtd\_status}

\textbf{Parameters}

- \texttt{index}: The index of the Node (“1” for the first Node, etc.)
- \texttt{node\_id}: The retrieved ID of the Node

Listing 4.2: Opening and closing a connection to a Node.

```
#include <libwrtd.h>

int main(void) {
    wrtd\_dev *wrtd;
    wrtd\_status status;
    \texttt{uint32\_t} node\_count;
    \texttt{uint32\_t} node\_id;

    /* Not really used in this example */
    status = wrtd\_get\_node\_count(&node\_count);

    /* Get the ID of the first Node */
    status = wrtd\_get\_node\_id(1, &node\_id);

    /* Access the first Node */
    status = wrtd\_init(node\_id, false, NULL, &wrtd);

    /* This will erase all defined rules and alarms from the Node, so it might not be what you want the program to do every time it is executed. */
    wrtd\_reset(wrtd);

    /* Do some more work here... */

    wrtd\_close(wrtd);
}
```
4.1.3 Attribute Handling API

The Attribute Handling API defines the available Attributes and the Functions for accessing them. Attributes can be of type bool, int32, string, or tstamp (wrtd_tstamp). Access can be RW (read/write), RO (read-only) or WO (write-only).

Furthermore they can be related to a specific alarm, rule, application, or they can be global (they apply to the whole device).

Attributes are accessed by means of the following functions, depending on their type:

- wrtd_set_attr_bool()
- wrtd_get_attr_bool()
- wrtd_set_attr_int32()
- wrtd_get_attr_int32()
- wrtd_set_attr_string()
- wrtd_get_attr_string()
- wrtd_set_attr_tstamp()
- wrtd_get_attr_tstamp()

When using one of the above functions to access a “global” Attribute, the Repeated Capability ID parameter should be set to WRDT_GLOBAL_REP_CAP_ID.

WRDT_GLOBAL_REP_CAP_ID
A repeated capability identifier for global attributes.

Listing 4.3: Accessing a “global” Attribute.

```c
#include <libwrtd.h>

int main(void) {  
    wrtd_dev *wrtd;
    wrtd_status status;
    bool log_empty;

    status = wrtd_init(1, false, NULL, &wrtd);

    status = wrtd_get_attr_bool(wrtd, WRDT_GLOBAL_REP_CAP_ID,  
                                WRDT_ATTR_EVENT_LOG_EMPTY,  
                                &log_empty);

    wrtd_close(wrtd);
}
```

Attributes

enum wrtd_attr
   White Rabbit Trigger Distribution attributes.
Names and values inspired by IVI-3.2 and IVI-3.15.

Values:

`__WRTD_ATTR_BASE = 1150000`
Same as `IVI_INSTR_SPECIFIC_ATTR_BASE`.

`WRTD_ATTR_EVENT_LOG_EMPTY = __WRTD_ATTR_BASE + 0x00`
RO bool global True if the Event Log is empty.

`WRTD_ATTR_EVENT_LOG_ENABLED = __WRTD_ATTR_BASE + 0x01`
RW bool global Enable/disable the Event Log.

`WRTD_ATTR_IS_TIME_SYNCHRONIZED = __WRTD_ATTR_BASE + 0x02`
RO bool global True if the device is synchronized to White Rabbit time.

`WRTD_ATTR_SYS_TIME = __WRTD_ATTR_BASE + 0x03`
RO tstamp global Current system time.

`WRTD_ATTR_ALARM_COUNT = __WRTD_ATTR_BASE + 0x10`
RO int32 global Number of defined Alarms.

`WRTD_ATTR_ALARM_ENABLED = __WRTD_ATTR_BASE + 0x11`
RW bool alarm Enable/disable an Alarm.

`WRTD_ATTR_ALARM_SETUP_TIME = __WRTD_ATTR_BASE + 0x12`
RW tstamp alarm Specifies at what time to send an Alarm event. This is typically set to a moment earlier than `WRTD_ATTR_ALARM_TIME`, to allow for the event to reach its destination(s) before the `WRTD_ATTR_ALARM_TIME` moment.

`WRTD_ATTR_ALARM_TIME = __WRTD_ATTR_BASE + 0x13`
RW tstamp alarm Specifies at what time to trigger an Alarm.

`WRTD_ATTR_ALARM.PERIOD = __WRTD_ATTR_BASE + 0x14`
RW tstamp alarm Specifies the Alarm period. 0 means no repetitions.

`WRTD_ATTR_ALARM_REPEAT_COUNT = __WRTD_ATTR_BASE + 0x15`
RW int32 alarm Specifies the number of times an Alarm will occur at the period specified by `WRTD_ATTR_ALARM.PERIOD`, before becoming automatically disabled. 0 means infinite repetitions. 1 means that the alarm will occur only once. When read, it returns the remaining repetitions.

`WRTD_ATTR_RULE_COUNT = __WRTD_ATTR_BASE + 0x20`
RO int32 global Number of defined Rules.

`WRTD_ATTR_RULE_ENABLED = __WRTD_ATTR_BASE + 0x21`
RW bool rule Enable/disable a Rule.

`WRTD_ATTR_RULE_REPEAT_COUNT = __WRTD_ATTR_BASE + 0x22`
RW int32 rule Specifies the number of times a Rule will fire before becoming automatically disabled. 0 means infinite. 1 means that the Rule will fire only once. When read, it returns the remaining repetitions.

`WRTD_ATTR_RULE_SOURCE = __WRTD_ATTR_BASE + 0x23`
RW string rule Get/set Rule source. Rule sources can be:

- Local source Event IDs (in the form of LC-I<x>)
- Alarm IDs (any ID with an `alarm` prefix)
- Any other string which will be interpreted as a network message Event ID.
WRTD_ATTR_RULE_DESTINATION = __WRTD_ATTR_BASE + 0x24
RW string rule Get/set Rule destinations. Rule destinations can be:
  - Local destination Event IDs (in the form of LC-O<x>)
  - Any other string which will be interpreted as a network message Event ID.

WRTD_ATTR_RULE_SEND_LATE = __WRTD_ATTR_BASE + 0x25
RW bool rule If true, events that arrive late (with a timestamp in the past) will still be executed; otherwise they are logged and discarded.

WRTD_ATTR_RULE_DELAY = __WRTD_ATTR_BASE + 0x26
RW tstamp rule Specifies the delay to add to the timestamp of the source Event before forwarding it to its destination. The provided value must be less than 1 second.

WRTD_ATTR_RULE_HOLDOFF = __WRTD_ATTR_BASE + 0x27
RW tstamp rule Specifies the “dead time” between two source Events. Any new event that arrives during this time, will be logged and discarded. The provided value must be less than 1 second.

WRTD_ATTR_RULE_RESYNC_PERIOD = __WRTD_ATTR_BASE + 0x28
RW tstamp rule Re-align the source Event timestamp to a multiple of this value. As an example, if the Event timestamp is 00:00:12.000.123.456 and RESYNC_PERIOD is 00:00:00.000.005.000 (5us), the Event timestamp will be re-aligned to 00:00:12.000.125.000. This calculation is done after applying the delay defined by WRTD_ATTR_RULE_DELAY. The provided value must be less than 1 second.

WRTD_ATTR_RULE_RESYNC_FACTOR = __WRTD_ATTR_BASE + 0x29
RW int32 rule Further re-align the source Event timestamp to a multiple of WRTD_ATTR_RULE_RESYNC_PERIOD. As an example, if RESYNC_PERIOD is 5us and the provided value is 10, 50us will be added to the Event timestamp, after applying WRTD_ATTR_RULE_DELAY and re-aligning it to the RESYNC_PERIOD.

WRTD_ATTR_STAT_RULE_RX_EVENTS = __WRTD_ATTR_BASE + 0x30
RO int32 rule Number of received Events for a Rule.

WRTD_ATTR_STAT_RULE_RX_LAST = __WRTD_ATTR_BASE + 0x31
RO tstamp rule Timestamp of most recently received Event for a Rule.

WRTD_ATTR_STAT_RULE_TX_EVENTS = __WRTD_ATTR_BASE + 0x32
RO int32 rule Number of transmitted Events for a Rule.

WRTD_ATTR_STAT_RULE_TX_LAST = __WRTD_ATTR_BASE + 0x33
RO tstamp rule Timestamp of most recently transmitted Event for a Rule.

WRTD_ATTR_STAT_RULE_MISSED_LATE = __WRTD_ATTR_BASE + 0x34
RO int32 rule Number of received Events that arrived too late. See also WRTD_ATTR_RULE_SEND_LATE.

WRTD_ATTR_STAT_RULE_MISSED_HOLDOFF = __WRTD_ATTR_BASE + 0x35
RO int32 rule Number of received Events that arrived during hold-off. See also WRTD_ATTR_RULE_HOLDOFF.

WRTD_ATTR_STAT_RULE_MISSED_NOSYNC = __WRTD_ATTR_BASE + 0x36
RO int32 rule Number of received Events that were discarded because the device was not synchronized to White Rabbit. See also WRTD_ATTR_IS_TIME_SYNCHRONIZED.
WRTD_ATTR_STAT_RULE_MISSED_OVERFLOW = __WRTD_ATTR_BASE + 0x37
RO int32 rule Number of received Events that were discarded because of internal buffer overflows. This may happen if the Event rate is too high.

WRTD_ATTR_STAT_RULE_MISSED_LAST = __WRTD_ATTR_BASE + 0x38
RO tstamp rule Timestamp of most recently missed Event.

WRTD_ATTR_STAT_RULE_RX_LATENCY_MIN = __WRTD_ATTR_BASE + 0x39
RO tstamp rule Minimum latency between the Event timestamp and its reception by WRTD.

WRTD_ATTR_STAT_RULE_RX_LATENCY_MAX = __WRTD_ATTR_BASE + 0x3A
RO tstamp rule Maximum latency between the Event timestamp and its reception by WRTD.

WRTD_ATTR_STAT_RULE_RX_LATENCY_AVG = __WRTD_ATTR_BASE + 0x3B
RO tstamp rule Average latency between the Event timestamp and its reception by WRTD.

WRTD_ATTR_FW_COUNT = __WRTD_ATTR_BASE + 0x80
RO int32 global Number of separate WRTD firmware applications running on the device.

WRTD_ATTR_FW_MAJOR_VERSION = __WRTD_ATTR_BASE + 0x81
RO int32 application Major part of the firmware’s version.

WRTD_ATTR_FW_MINOR_VERSION = __WRTD_ATTR_BASE + 0x82
RO int32 application Major part of the firmware’s version.

WRTD_ATTR_FW_MAJOR_VERSION_REQUIRED = __WRTD_ATTR_BASE + 0x83
RO int32 application Major part of WRTD version supported by the firmware. It must be equal to the major version of the WRTD library in use.

WRTD_ATTR_FW_MINOR_VERSION_REQUIRED = __WRTD_ATTR_BASE + 0x84
RO int32 application Minor part of WRTD version supported by the firmware. It must be less than or equal to the minor version of the WRTD library in use.

WRTD_ATTR_FW_MAX_RULES = __WRTD_ATTR_BASE + 0x85
RO int32 application Max number of rules allowed by this firmware.

WRTD_ATTR_FW_MAX_ALARMS = __WRTD_ATTR_BASE + 0x86
RO int32 application Max number of alarms allowed by this firmware.

WRTD_ATTR_FW_CAPABILITIES = __WRTD_ATTR_BASE + 0x88
RO int32 application Event TX/RX capabilities of this firmware. This is a bit-field with the following meaning per bit:

- bit 0: if set, the firmware can receive network events.
- bit 1: if set, the firmware can create network events.
- bit 2: if set, the firmware can receive local events.
- bit 3: if set, the firmware can create local events.

WRTD_ATTR_FW_LOCAL_INPUTS = __WRTD_ATTR_BASE + 0x8A
RO int32 application Number of available local input channels.

WRTD_ATTR_FW_LOCAL_OUTPUTS = __WRTD_ATTR_BASE + 0x8B
RO int32 application Number of available local output channels.
__WRTD_ATTR_MAX_NUMBER
Always last entry in this enum

Functions

wrt_status wrtd_set_attr_bool(wrtd_dev *wrtd, const char *rep_cap_id, wrtd_attr id, bool value)
Set an attribute of type bool.
Modelled after the IVI-C SetAttribute family of functions.

Return wrt_status

Parameters

• wrtd: Device token.
• rep_cap_id: ID (string) of concerned repeated capability. If it is a global attribute, use WRTD_GLOBAL_REP_CAP_ID
• id: ID (wrt_attr) of concerned attribute.
• value: Value to write to the attribute.

wrt_status wrtd_get_attr_bool(wrtd_dev *wrtd, const char *rep_cap_id, wrtd_attr id, bool *value)
Get an attribute of type bool.
Modelled after the IVI-C GetAttribute family of functions.

Return wrt_status

Parameters

• wrtd: Device token.
• rep_cap_id: ID (string) of concerned repeated capability. If it is a global attribute, use WRTD_GLOBAL_REP_CAP_ID
• id: ID (wrt_attr) of concerned attribute.
• value: Retrieved attribute value.

wrt_status wrtd_set_attr_int32(wrtd_dev *wrtd, const char *rep_cap_id, wrtd_attr id, int32_t value)
Set an attribute of type int32.
Modelled after the IVI-C SetAttribute family of functions.

Return wrt_status

Parameters

• wrtd: Device token.
• rep_cap_id: ID (string) of concerned repeated capability. If it is a global attribute, use WRTD_GLOBAL_REP_CAP_ID
• id: ID (wrt_attr) of concerned attribute.
• value: Value to write to the attribute.

```c
wrtd_status wrtd_get_attr_int32 (wrtd_dev *wrtd, const char *rep_cap_id, wrtd_attr id, int32_t *value)
```

Get an attribute of type `int32_t`.

Modelled after the IVI-C GetAttribute family of functions.

**Return** `wrtd_status`

**Parameters**

• `wrtd`: Device token.

• `rep_cap_id`: ID (string) of concerned repeated capability. If it is a global attribute, use `WRTD_GLOBAL_REP_CAP_ID`.

• `id`: ID (`wrtd_attr`) of concerned attribute.

• `value`: Retrieved attribute value.

```c
wrtd_status wrtd_set_attr_string (wrtd_dev *wrtd, const char *rep_cap_id, wrtd_attr id, const char *value)
```

Set an attribute of type `string`.

Modelled after the IVI-C SetAttribute family of functions.

**Return** `wrtd_status`

**Parameters**

• `wrtd`: Device token.

• `rep_cap_id`: ID (string) of concerned repeated capability. If it is a global attribute, use `WRTD_GLOBAL_REP_CAP_ID`.

• `id`: ID (`wrtd_attr`) of concerned attribute.

• `value`: Value to write to the attribute.

```c
wrtd_status wrtd_get_attr_string (wrtd_dev *wrtd, const char *rep_cap_id, wrtd_attr id, int32_t value_buf_size, char *value)
```

Get an attribute of type `string`.

Modelled after the IVI-C GetAttribute family of functions.

This function complies with IVI-3.2 section, 3.1.2.1 (Additional Compliance Rules for C Functions with ViChar Array Output Parameters), with the exception of the buffer_size < 0 case, which produces an error instead of allowing a potential buffer overflow.

**Return** `wrtd_status`. However, if the buffer size parameter is 0, then this function returns instead a positive value, indicating the minimum buffer size necessary to fit the full message. See also IVI-3.2, section 3.1.2.1.

**Parameters**

• `wrtd`: Device token.

• `rep_cap_id`: ID (string) of concerned repeated capability. If it is a global attribute, use `WRTD_GLOBAL_REP_CAP_ID`
• id: ID (wrtd_attr) of concerned attribute.
• value_buffer_size: Size of pre-allocated value buffer.
• value: Retrieved attribute value.

**wrtd_status** wrtd_set_attr_tstamp(wrtd_dev *wrtd, const char *rep_cap_id, wrtd_attr id, const wrtd_tstamp *value)

Set an attribute of type timestamp.

Modelled after the IVI-C SetAttribute family of functions.

**Return** wrtd_status

**Parameters**

• wrtd: Device token.
• rep_cap_id: ID (string) of concerned repeated capability. If it is a global attribute, use WRTD_GLOBAL_REP_CAP_ID
• id: ID (wrtd_attr) of concerned attribute.
• value: Value to write to the attribute.

**wrtd_status** wrtd_get_attr_tstamp(wrtd_dev *wrtd, const char *rep_cap_id, wrtd_attr id, wrtd_tstamp *value)

Get an attribute of type timestamp.

Modelled after the IVI-C GetAttribute family of functions.

**Return** wrtd_status

**Parameters**

• wrtd: Device token.
• rep_cap_id: ID (string) of concerned repeated capability. If it is a global attribute, use WRTD_GLOBAL_REP_CAP_ID
• id: ID (wrtd_attr) of concerned attribute.
• value: Retrieved attribute value.

Listing 4.4: Accessing various types of attributes.

```c
#include <libwrtd.h>

int main(void) {
    wrtd_dev *wrtd;
    wrtd_status status;
    wrtd_tstamp ts;
    bool log_empty;

    status = wrtd_init(1, false, NULL, &wrtd);

    /* check if the event log is empty (global attribute) */
    status = wrtd_get_attr_bool(wrtd, WRTD_GLOBAL_REP_CAP_ID,
```
/* add a rule with name "rule1" */
status = wrtd_add_rule(wrtd, "rule1");

/* set the repeat count for "rule1" */
status = wrtd_set_attr_int32(wrtd, "rule1",
WRTD_ATTR_RULE_REPEAT_COUNT, 5);

/* set the source for "rule1" to local channel input 1 */
status = wrtd_set_attr_string(wrtd, "rule1",
WRTD_ATTR_RULE_SOURCE, "LC-I1");

/* get the delay configured for "rule1" */
status = wrtd_get_attr_tstamp(wrtd, "rule1",
WRTD_ATTR_RULE_DELAY, &ts);

wrtd_close(wrtd);

---

Important: wrtd_get_attr_tstamp() and wrtd_set_attr_tstamp() allow getting and
setting of timestamp attributes. Within the C library, a timestamp is represented as a C struct:

struct wrtd_tstamp
  WRTD timestamp format

Public Members

uint32_t seconds
  TAI seconds since 1/Jan/1970 (Unix Epoch Time). This will overflow in
  7/Feb/2106...

uint32_t ns
  Number of nanoseconds. Wraps at 1e9.

uint32_t frac
  Number of fractional nanoseconds. Unit is $2^{-32}$ ns.

Note that the above internal representation is slightly different than the official definition of a
timestamp. In particular, the seconds counter is 16-bits shorter and the fractional nanosecond
counter is 16 bits longer (and every “tick” represents $2^{32}$ns). Both of these changes help the
underlying firmware to operate faster. When WRTD sends (or receives) a message, it will always
use the official definition of a timestamp and convert it automatically to the above internal
representation when necessary.
4.1.4 Event Logging API

The Event Logging API provides functions for accessing the Event Log.

wrtd_status wrtd_clear_event_log_entries (wrtd_dev *wrd)
Clear all entries from the Event Log.
Modelled after the IVI-C ClearEventLog function (from IVI-3.15).

Return wrtd_status.

Parameters

• wrtd: Device token.

wrtd_status wrtd_get_next_event_log_entry (wrtd_dev *wrd, int32_t log_entry_buffer_size, char *log_entry)
Retrieve and clear the last entry from the Event Log.
Modelled after the IVI-C GetNextEventLogEntry function (from IVI-3.15).
This function complies with IVI-3.2, section 3.1.2.1 (Additional Compliance Rules for C Functions with ViChar Array Output Parameters), with the exception of the buffer_size < 0 case, which produces an error instead of allowing a potential buffer overflow.

Event log entries use the format id|seq|log tstamp|event tstamp|log type|log reason, where:

• id is the Event ID.
• seq is a sequence number for the rule that generated this Event. Each Event generated by the Rule causes the sequence number to increase by one. Limited to 4 digits.
• log tstamp is the timestamp of when the event was logged.
• event tstamp is the timestamp of when the event happened.
• log type is the type of the event.
• log reason is the specific reason for this type of event.

All event log entries have a fixed length of WRTD_LOG_ENTRY_SIZE.

Timestamps are represented using the format YYYY-MM-DD, hh:mm:ss.mmm.uuu.nnn+fff, where:

• YYYY-MM-DD is the year, month and day.
• hh:mm:ss is hours, minutes and seconds.
• mmm.uuu.nnn is milliseconds, microseconds and nanoseconds.
• fff is fractional nanoseconds. The unit is 2e-9 nanoseconds.

Log types and reasons can be one of the following (in the form type|reason):

• GENERATED|ALARM: An Event was generated because of an Alarm.
• GENERATED|DEVICE_x: An Event was generated from device x.
• CONSUMED|START: An Event with a local channel output as destination has been forwarded and scheduled to execute. The exact meaning of this log type is application-specific.
- **CONSUMED|DONE**: An Event with a local channel output as destination has finished executing. The exact meaning of this log type is application-specific.

- **DISCARDED|NO SYNC**: An Event was discarded because the device was not synchronized to White Rabbit. See also `WRTD_ATTR_STAT_RULE_MISSED_NOSYNC`.

- **DISCARDED|HOLD OFF**: An Event was discarded because it violated the programmed hold-off value. See also `WRTD_ATTR_STAT_RULE_MISSED_HOLDOFF`.

- **DISCARDED|TIME OUT**: An Event was discarded because it arrived too late. See also `WRTD_ATTR_STAT_RULE_MISSED_LATE`.

- **DISCARDED|OVERFLOW**: An Event was discarded because of internal buffer overflows. This may happen if the Event rate is too high. See also `WRTD_ATTR_STAT_RULE_MISSED_OVERFLOW`.

- **NETWORK|TX**: An Event was sent over the White Rabbit network.

- **NETWORK|RX**: An Event was received over the White Rabbit network.

**Return** `wrtd_status`. However, if the buffer size parameter is 0, then this function returns instead a positive value, indicating the minimum buffer size necessary to fit the full message. See also IVI-3.2, section 3.1.2.1.

**Parameters**

- `wrtd`: Device token.
- `log_entry_buffer_size`: Size of pre-allocated `log_entry` buffer.
- `log_entry`: Buffer to store the log message string.

**Listing 4.5: Accessing the Event Log.**

```c
#include <libwrtd.h>

int main(void) {
    wrtd_dev *wrtd;
    char *log_msg;
    int buf_size;

    status = wrtd_init(1, false, NULL, &wrtd);

    /* clear the event log */
    status = wrtd_clean_event_log_entries(wrtd);

    /* query the size of the next event log message */
    buf_size = wrtd_get_next_event_log_entry(wrtd, 0, NULL);

    /* allocate the buffer for the log message */
    log_msg = malloc(sizeof(char), buf_size);

    /* retrieve the next event log message */
    status = wrtd_get_next_event_log_entry(wrtd, buf_size, log_msg);

    return status;
}
```

(continues on next page)
wrtld_close(wrtd);

return 0;
}

Hint: If you want to be sure that the buffer that you pass to
wrtld_get_next_event_log_entry() is large enough, without having to resort to query-
ing like in Listing 4.5, you can always allocate a buffer of \texttt{WRTD_LOG_ENTRY_SIZE}. WRTD
 guarantees that all event log entries shall not exceed this size.

\textbf{WRTD_LOG_ENTRY_SIZE}

Size (in characters, including null termination) of an event log enty.

Listing 4.6: Accessing the Event Log with a pre-defined buffer
size

\begin{verbatim}
#include <libwrtd.h>

int main(void) {
    wrtd_dev *wrtd;
    char log_msg[WRTD_LOG_ENTRY_SIZE];
    status = wrtd_init(1, false, NULL, &wrtd);

    /* retrieve the next event log message */
    status = wrtd_get_next_event_log_entry(wrtd, WRTD_LOG_ENTRY_SIZE, log_msg);

    wrtd_close(wrtd);

    return 0;
}
\end{verbatim}

4.1.5 Alarms API

The Alarms API provides functions for adding, removing and indexing Alarms.
Configuration of an Alarm happens by setting the relevant Attributes via the Attribute Handling API.

\texttt{wrtld_status} \texttt{wrtld_add_alarm}(wrtld_dev *wrtd, const char *rep_cap_id)

Create a new Alarm.

Complies with the IVI “Parameter style” for repeated capabilities (see IVI-3.4, section 12).

Return \texttt{wrtld_status}

Parameters

- \texttt{wrtd}: Device token.
• `rep_cap_id`: Name for the new Alarm. Must begin with ‘alarm’ or ‘ALARM’.

`wrtd_status wrtd_disable_all_alarms (wrtd_dev *wrtd)`  
Disable all defined Alarms.

Complies with the IVI “Parameter style” for repeated capabilities (see IVI-3.4, section 12).

See also, `WRTD_ATTR_ALARM_ENABLED` and `wrtd_set_attr_bool` for disabling a single Alarm.

**Return** `wrtd_status`

**Parameters**

• `wrtd`: Device token.

`wrtd_status wrtd_remove_alarm (wrtd_dev *wrtd, const char *rep_cap_id)`  
Remove an Alarm. The Alarm must not be enabled.

Complies with the IVI “Parameter style” for repeated capabilities (see IVI-3.4, section 12).

**Return** `wrtd_status`

**Parameters**

• `wrtd`: Device token.

• `rep_cap_id`: Name of the Alarm to remove.

`wrtd_status wrtd_remove_all_alarms (wrtd_dev *wrtd)`  
Remove all defined Alarms. The Alarms must not be enabled.

Complies with the IVI “Parameter style” for repeated capabilities (see IVI-3.4, section 12).

**Return** `wrtd_status`

**Parameters**

• `wrtd`: Device token.

`wrtd_status wrtd_get_alarm_name (wrtd_dev *wrtd, int32_t index, int32_t name_buffer_size, char *name)`  
Retrieve the name of an Alarm, based on the provided index.

Complies with the IVI “Parameter style” for repeated capabilities (see IVI-3.4, section 12).

This function complies with IVI-3.2, section 3.1.2.1 (Additional Compliance Rules for C Functions with ViChar Array Output Parameters), with the exception of the `buffer_size < 0` case, which produces an error instead of allowing a potential buffer overflow.

**Return** `wrtd_status`. See also IVI-3.2, section 3.1.2.1.

**Parameters**

• `wrtd`: Device token.

• `index`: index of the Alarm.

• `name_buffer_size`: Size of pre-allocated name buffer.
• name: Buffer to store the retrieved Alarm name.

Listing 4.7: Adding, removing and indexing Alarms.

```c
#include <libwrtd.h>

int main(void) {
    int i, count;
    char rep_cap_id[16];
    wrtd_dev *wrtd;

    status = wrtd_init(1, false, NULL, &wrtd);

    /* disable and then remove any declared alarms */
    status = wrtd_disable_all_alarms(wrtd);
    status = wrtd_remove_all_alarms(wrtd);

    /* Add three alarms */
    status = wrtd_add_alarm(wrtd, "alarm1");
    status = wrtd_add_alarm(wrtd, "alarm2");
    status = wrtd_add_alarm(wrtd, "alarm3");

    /* Remove the 2nd alarm */
    status = wrtd_remove_alarm(wrtd, "alarm2");

    /* Get number of defined alarms */
    status = wrtd_get_attr_int32(wrtd, WRTD_GLOBAL_REP_CAP_ID,
                                WRTD_ATTR_ALARM_COUNT, &count);

    /* Iterate through alarms and print their names.
     * This should output:
     * 1: alarm1
     * 2: alarm3
     */
    for (i = 0; i < count; i++) {
        status = wrtd_get_alarm_name(wrtd, i, 16, rep_cap_id);
        printf("%d: %s\n", i, rep_cap_id);
    }

    wrtd_close(wrtd);

    return 0;
}
```

4.1.6 Rules API

The Rules API provides functions for adding, removing and indexing Rules.
Configuration of a Rule happens by setting the relevant Attributes via the Attribute Handling API.
**wrtd_status** `wrtd_add_rule(wrtd_dev *wrtd, const char *rep_cap_id)`
Create a new Rule.

Complies with the IVI “Parameter style” for repeated capabilities (see IVI-3.4, section 12).

**Return** `wrtd_status`

**Parameters**
- `wrtd`: Device token.
- `rep_cap_id`: Name for the new Rule.

**wrtd_status** `wrtd_disable_all_rules(wrtd_dev *wrtd)`
Disable all defined Rules.

Complies with the IVI “Parameter style” for repeated capabilities (see IVI-3.4, section 12).

See also, `WRTD_ATTR_RULE_ENABLED` and `wrtd_set_attr_bool` for disabling a single Rule.

**Return** `wrtd_status`

**Parameters**
- `wrtd`: Device token.

**wrtd_status** `wrtd_remove_rule(wrtd_dev *wrtd, const char *rep_cap_id)`
Remove a Rule. The Rule must not be enabled.

Complies with the IVI “Parameter style” for repeated capabilities (see IVI-3.4, section 12).

**Return** `wrtd_status`

**Parameters**
- `wrtd`: Device token.
- `rep_cap_id`: Name of the Rule to remove.

**wrtd_status** `wrtd_remove_all_rules(wrtd_dev *wrtd)`
Remove all defined Rules. The Rules must not be enabled.

Complies with the IVI “Parameter style” for repeated capabilities (see IVI-3.4, section 12).

**Return** `wrtd_status`

**Parameters**
- `wrtd`: Device token.

**wrtd_status** `wrtd_get_rule_name(wrtd_dev *wrtd, int32_t index, int32_t name_buffer_size, char *name)`
Retrieve the name of a Rule, based on the provided index.

Complies with the IVI “Parameter style” for repeated capabilities (see IVI-3.4, section 12).

This function complies with IVI-3.2, section 3.1.2.1 (Additional Compliance Rules for C Functions with ViChar Array Output Parameters), with the exception of the buffer_size < 0 case, which produces an error instead of allowing a potential buffer overflow.
Return \textbf{wrtd\_status}. See also IVI-3.2, section 3.1.2.1.

Parameters

- \textit{wrtd}: Device token.
- \textit{index}: index of the Rule.
- \textit{name\_buffer\_size}: Size of pre-allocated name buffer.
- \textit{name}: Buffer to store the retrieved Rule name.

\textit{wrtd\_status} \textbf{wrtd\_reset\_rule\_stats} (\texttt{wrtdev *wrtd, const char *rep\_cap\_id})

Reset all statistics for the given rule. The Rule must not be enabled.

Complies with the IVI “Parameter style” for repeated capabilities (see IVI-3.4, section 12).

Return \textit{wrtd\_status}

Parameters

- \textit{wrtd}: Device token.
- \textit{rep\_cap\_id}: Name of the Rule to reset its statistics.

Listing 4.8: Adding, removing and indexing \textit{Rules}.

```c
#include <libwrtdev.h>

int main(void) {
    int i, count;
    char rep_cap_id[16];
    wrtd_dev *wrtdev;

    status = wrtd_init(1, false, NULL, &wrtdev);

    /* disable and then remove any declared rules */
    status = wrtd_disable_all_rules(wrtdev);
    status = wrtd_remove_all_rules(wrtdev);

    /* Add three rules */
    status = wrtd_add_rule(wrtdev, "rule1");
    status = wrtd_add_rule(wrtdev, "rule2");
    status = wrtd_add_rule(wrtdev, "rule3");

    /* Remove the 2nd rule */
    status = wrtd_remove_rule(wrtdev, "rule2");

    /* Get number of defined rules */
    status = wrtd_get_attr_int32(wrtdev, WRTD_GLOBAL_REP_CAP_ID, WRTD_ATTR_RULE_COUNT, &count);

    /* Iterate through rules and print their names.
     * This should output:
     * 1: rule1
     * */
    (continues on next page)
```
2: rule3
*/
for (i = 0; i < count; i++) {
    status = wrtd_get_rule_name(wrtd, i, 16, rep_cap_id);
    printf ("%d: %s\n", i, rep_cap_id);
}

wrtd_close(wrtd);

Listing 4.9: Basic Rule configuration.

#include <libwrtd.h>

int main(void) {
    wrtd_dev *wrtd;

    status = wrtd_init(1, false, NULL, &wrtd);

    /* Add a rule */
    status = wrtd_add_rule(wrtd, "rule1");

    /* Set rule to listen for events coming on local channel input 1 and generate a message on the network with event ID "NET0", after adding 500ns to the event timestamp. */
    status = wrtd_set_attr_string(wrtd, "rule1",
        WRTD_ATTR_RULE_SOURCE, "LC-I1");
    status = wrtd_set_attr_string(wrtd, "rule1",
        WRTD_ATTR_RULE_DESTINATION, "NET0");
    wrtd_tstamp ts = {.seconds = 0, .ns = 500, .frac = 0};
    status = wrtd_set_attr_tstamp(wrtd, "rule1",
        WRTD_ATTR_RULE_DELAY, &ts);

    /* Enable rule */
    status = wrtd_set_attr_bool(wrtd, "rule1",
        WRTD_ATTR_RULE_ENABLED, True);

    wrtd_close(wrtd);

    return 0;
}

4.1.7 Applications API

Similar to Rules and Alarms, Applications are also Repeated Capabilities. However, they are read-only and, as such, do not possess any functions to add or remove them. The only provided functionality is that of indexing.
Information retrieval regarding a particular Application is performed by by getting the relevant Attributes via the Attribute Handling API.

```c
wrtd_status wrtd_get_fw_name(wrtd_dev *wrtd, int32_t index, int32_t name_buffer_size, char *name)
```

Retrieve the name of a firmware application, based on the provided index.

Complies with the IVI “Parameter style” for repeated capabilities (see IVI-3.4, section 12).

This function complies with IVI-3.2, section 3.1.2.1 (Additional Compliance Rules for C Functions with ViChar Array Output Parameters), with the exception of the buffer_size < 0 case, which produces an error instead of allowing a potential buffer overflow.

Return `wrtd_status`. See also IVI-3.2, section 3.1.2.1.

Parameters

- `wrtd`: Device token.
- `index`: index of the firmware application.
- `name_buffer_size`: Size of pre-allocated name buffer.
- `name`: Buffer to store the retrieved application name.

Listing 4.10: Indexing Applications and version retrieval.

```c
#include <libwrtd.h>

int main(void) {
    int i, count, major, minor;
    char rep_cap_id[16];
    wrtd_dev *wrtd;

    status = wrtd_init(1, false, NULL, &wrtd);

    /* Get number of defined applications */
    status = wrtd_get_attr_int32(wrtd, WRTD_GLOBAL_REP_CAP_ID, WRTD_ATTR_FW_COUNT, &count);

    /* Iterate through applications and print their names and versions. */
    for (i = 0; i < count; i++) {
        status = wrtd_get_fw_name(wrtd, i, 16, rep_cap_id);
        status = wrtd_get_attr_int32(wrtd, rep_cap_id, WRTD_ATTR_FW_MAJOR_VERSION, &major);
        status = wrtd_get_attr_int32(wrtd, rep_cap_id, WRTD_ATTR_FW_MINOR_VERSION, &minor);
        printf("%d: %s, v%d.%d\n", i, rep_cap_id, major, minor);
    }

    wrtd_close(wrtd);
}
```

(continues on next page)
4.2 Python Wrapper

The WRTD Python wrapper provides a thin wrapper around the C Library, using the Python ctypes package.

The wrapper is provided as a Python package with a single class that encapsulates the complete WRTD C Library:

```python
class PyWrtld.

    def __init__(self, node_id):
        self.node_id = node_id

    def get_node_count(self):
        return 0;

    def get_node_id(self):
        return 0;

4.2.1 Initialisation API

To start using it, simply import the PyWrtld package and instantiate a PyWrtld object, passing to it the ID of the Node you wish to access.

In order to retrieve the ID of the Node, the PyWrtld class provides the static methods PyWrtld.
PyWrtld.get_node_count() and PyWrtld.PyWrtld.get_node_id() that can be used before you instantiate the PyWrtld object.

If the ID is wrong or if the user does not have the correct permissions to access it, WRTD will return WRTD_ERRORRESOURCE_UNKNOWN.

```
static PyWrtd.get_node_count()
Corresponds to C library wrtd_get_node_count().

static PyWrtd.get_node_id(index)
Corresponds to C library wrtd_get_node_id().

PyWrtd.reset()
Corresponds to C library wrtd_reset().

4.2.2 Attribute Handling API

Getting and setting boolean, integer and string Attributes is straight-forward.

Boolean Attributes

PyWrtd.set_attr_bool(rep_cap_id, id, value)
Corresponds to C library wrtd_set_attr_bool().

Parameters

• rep_cap_id – Repeated Capability ID
• id – ID of concerned Attribute
• value – Value to write to the Attribute

PyWrtd.get_attr_bool(rep_cap_id, id)
Corresponds to C library wrtd_get_attr_bool().

Parameters

• rep_cap_id – Repeated Capability ID
• id – ID of concerned Attribute

Returns Retrieved attribute value

```python
>>> wrtd.add_alarm('alarm1')
>>> wrtd.get_attr_bool('alarm1', PyWrtd.WRTD_ATTR_ALARM_ENABLED)
False
>>> wrtd.set_attr_bool('alarm1', PyWrtd.WRTD_ATTR_ALARM_ENABLED, True)
>>> wrtd.get_attr_bool('alarm1', PyWrtd.WRTD_ATTR_ALARM_ENABLED)
True
```
Integer Attributes

PyWrtd.set_attr_int32(rep_cap_id, id, value)
Corresponds to C library wrtd_set_attr_int32().

Parameters

- rep_cap_id – Repeated Capability ID
- id – ID of concerned Attribute
- value – Value to write to the Attribute

PyWrtd.get_attr_int32(rep_cap_id, id)
Corresponds to C library wrtd_get_attr_int32().

Parameters

- rep_cap_id – Repeated Capability ID
- id – ID of concerned Attribute

Returns Retrieved attribute value

```python
>>> wrtd.add_alarm('alarm1')
>>> wrtd.get_attr_int32('alarm1', PyWrtd.WRTD_ATTR_ALARM_REPEAT_COUNT)
0
>>> wrtd.set_attr_int32('alarm1', PyWrtd.WRTD_ATTR_ALARM_REPEAT_COUNT, 5)
>>> wrtd.get_attr_int32('alarm1', PyWrtd.WRTD_ATTR_ALARM_REPEAT_COUNT)
5
```

String Attributes

PyWrtd.set_attr_string(rep_cap_id, id, value)
Corresponds to C library wrtd_set_attr_string().

Parameters

- rep_cap_id – Repeated Capability ID
- id – ID of concerned Attribute
- value – Value to write to the Attribute

PyWrtd.get_attr_string(rep_cap_id, id)
Corresponds to C library wrtd_get_attr_string().

Parameters

- rep_cap_id – Repeated Capability ID
- id – ID of concerned Attribute

Returns Retrieved attribute value

```python
>>> wrtd.add_rule('rule1')
>>> wrtd.set_attr_string('rule1', PyWrtd.WRTD_ATTR_RULE_SOURCE, 'event0')
>>> wrtd.get_attr_string('rule1', PyWrtd.WRTD_ATTR_RULE_SOURCE)
'event0'
```
**Timestamp Attributes**

Getting and setting *Event Timestamp Attributes* is slightly different than in the *C Library*. The set method takes the `seconds`, `ns` and `frac` fields of the *Event Timestamp* as separate parameters.

PyWrtd.

set_attr_tstamp(rep_cap_id, id, seconds=0, ns=0, frac=0)

Corresponds to C library `wrdt_set_attr_tstamp()`.

**Parameters**

- `rep_cap_id` – *Repeated Capability ID*
- `id` – ID of concerned *Attribute*
- `seconds` – Seconds value to write to the *Attribute*
- `ns` – Nanoseconds value to write to the *Attribute*
- `frac` – Fractional nanoseconds value to write to the *Attribute*

The get method returns a Python dictionary with the `seconds`, `ns` and `frac` fields of the *Event Timestamp*.

PyWrtd.

get_attr_tstamp(rep_cap_id, id)

Corresponds to C library `wrdt_get_attr_tstamp()`.

**Parameters**

- `rep_cap_id` – *Repeated Capability ID*
- `id` – ID of concerned *Attribute*

**Returns** Retrieved attribute value (Python dictionary with `seconds`, `ns` and `frac` keys)

```python
>>> wrtd.add_alarm('alarm1')
>>> wrtd.get_attr_tstamp('alarm1', PyWrtd.WRTD_ATTR_ALARM_TIME)
{'frac': 0, 'ns': 0, 'seconds': 0}
>>> wrtd.set_attr_tstamp('alarm1', PyWrtd.WRTD_ATTR_ALARM_TIME, seconds=5, ns=40e3)
>>> wrtd.get_attr_tstamp('alarm1', PyWrtd.WRTD_ATTR_ALARM_TIME)
{'frac': 0, 'seconds': 5, 'ns': 40000}
```

**4.2.3 Event Logging API**

PyWrtd.

clear_event_log_entries()

Corresponds to C library `wrdt_clear_event_log_entries()`.

PyWrtd.

get_next_event_log_entry()

Corresponds to C library `wrdt_get_next_event_log_entry()`.

```python
>>> # First, enable logging
>>> wrtd.set_attr_bool(PyWrtd.WRTD_GLOBAL_REP_CAP_ID, PyWrtd.WRTD_ATTR_EVENT_LOG_ENABLED, True)
```

(continues on next page)
4.2.4 Alarms API

PyWrtd.add_alarm(rep_cap_id)

Corresponds to C library wrtd_add_alarm().

Parameters rep_cap_id – Repeated Capability ID of new Alarm

```python
>>> wrtd.get_attr_int32(PyWrtd.WRTD_GLOBAL_REP_CAP_ID, PyWrtd.WRTD_ATTR_ALARM_COUNT)
0
>>> wrtd.add_alarm('alarm1')
>>> wrtd.get_attr_int32(PyWrtd.WRTD_GLOBAL_REP_CAP_ID, PyWrtd.WRTD_ATTR_ALARM_COUNT)
1
```

PyWrtd.disable_all_alarms()

Corresponds to C library wrtd_disable_all_alarms().

```python
>>> wrtd.add_alarm('alarm1')
>>> wrtd.get_attr_bool('alarm1', PyWrtd.WRTD_ATTR_ALARM_ENABLED)
False
>>> wrtd.set_attr_bool('alarm1', PyWrtd.WRTD_ATTR_ALARM_ENABLED, True)
>>> wrtd.get_attr_bool('alarm1', PyWrtd.WRTD_ATTR_ALARM_ENABLED)
True
>>> wrtd.disable_all_alarms()
>>> wrtd.get_attr_bool('alarm1', PyWrtd.WRTD_ATTR_ALARM_ENABLED)
False
```

PyWrtd.remove_alarm(rep_cap_id)

Corresponds to C library wrtd_remove_alarm().

Parameters rep_cap_id – Repeated Capability ID of Alarm to remove

```python
>>> wrtd.add_alarm('alarm1')
>>> wrtd.get_attr_int32(PyWrtd.WRTD_GLOBAL_REP_CAP_ID, PyWrtd.WRTD_ATTR_ALARM_COUNT)
1
>>> wrtd.remove_alarm('alarm1')
>>> wrtd.get_attr_int32(PyWrtd.WRTD_GLOBAL_REP_CAP_ID, PyWrtd.WRTD_ATTR_ALARM_COUNT)
0
```

PyWrtd.remove_all_alarms()

Corresponds to C library wrtd_remove_all_alarms().
>>> wrtd.add_alarm('alarm1')
>>> count = wrtd.get_attr_int32(PyWrtd.WRTD_GLOBAL_REP_CAP_ID, PyWrtd.WRTD_ATTR_ALARM_COUNT)
1
>>> wrtd.remove_all_alarms()
>>> count = wrtd.get_attr_int32(PyWrtd.WRTD_GLOBAL_REP_CAP_ID, PyWrtd.WRTD_ATTR_ALARM_COUNT)
0

PyWrtd.get_alarm_name(index)
Corresponds to C library wrtd_get_alarm_name().

Parameters: index – Index of the Alarm

Returns: Repeated Capability ID of the Alarm

>>> wrtd.add_alarm('alarm1')
>>> for i in range(count):
...     wrtd.get_alarm_name(i)
'alarm1'

4.2.5 Rules API

PyWrtd.add_rule(rep_cap_id)
Corresponds to C library wrtd_add_rule().

Parameters: rep_cap_id – Repeated Capability ID of new Rule

>>> wrtd.add_rule('rule1')
>>> wrtd.get_attr_int32(PyWrtd.WRTD_GLOBAL_REP_CAP_ID, PyWrtd.WRTD_ATTR_RULE_COUNT)
0

PyWrtd.disable_all_rules()
Corresponds to C library wrtd_disable_all_alarms().

>>> wrtd.add_rule('rule1')
>>> wrtd.get_attr_bool('rule1', PyWrtd.WRTD_ATTR_RULE_ENABLED)
False
>>> wrtd.set_attr_bool('rule1', PyWrtd.WRTD_ATTR_RULE_ENABLED, True)
True
>>> wrtd.disable_all_rules()

(continues on next page)
PyWrtd.remove_rule(rep_cap_id)
    Corresponds to C library wrtd_remove_rule().

    Parameters rep_cap_id – Repeated Capability ID of Rule to remove

PyWrtd.remove_all_rules()
    Corresponds to C library wrtd_remove_all_rules().

PyWrtd.get_rule_name(index)
    Corresponds to C library wrtd_get_rule_name().

    Parameters index – Index of the Rule

    Returns Repeated Capability ID of the Rule

PyWrtd.reset_rule_stats(rep_cap_id)
    Corresponds to C library wrtd_reset_rule_stats().

    Parameters rep_cap_id – Repeated Capability ID of the Rule to reset its statistics
4.2.6 Applications API

PyWrtd.get_fw_name(index)

Corresponds to C library wrtd_get_fw_name().

Parameters  
index – Index of the Application

Returns  
Repeated Capability ID of the Application

# From the SVEC-based TDC+FDELAY reference Node

```python
>>> count = wrtd.get_attr_int32(PyWrtd.WRTD_GLOBAL_REP_CAP_ID, PyWrtd.WRTD_ATTR_FW_COUNT)
>>> print(count)
2
>>> for i in range(count):
...    wrtd.get_fw_name(i)
'wrtd-tdc'
'wrtd-fd'
```

4.3 Tools

WRTD provides a command-line, Python based tool (wrtd-tool) for accessing a Node.

Hint:  Please make sure that you run the tool with the proper permissions.

For details on how to install the tool (and their dependencies), please refer to Section 2.

4.3.1 wrtd-tool

wrtd-tool is a command-line tool that implements several different operations on a Node. It supports most of the functionality provided by the Python Wrapper.

A list of the available commands can be retrieved by passing the -h option to the tool:

```
> wrtd-tool -h
usage: wrtd-tool [-h] <command> ...
WRTD Node configuration tool
```
Optional arguments:

- `h`, `--help`    show this help message and exit

Available commands:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;command&gt;</td>
<td>(Use &quot;&lt;command&gt; -h&quot; to get more details)</td>
</tr>
<tr>
<td>list-nodes</td>
<td>List the IDs of all detected WRTD Nodes</td>
</tr>
<tr>
<td>sys-info</td>
<td>Show system information</td>
</tr>
<tr>
<td>sys-time</td>
<td>Show current system time</td>
</tr>
<tr>
<td>enable-log</td>
<td>Enable logging</td>
</tr>
<tr>
<td>disable-log</td>
<td>Disable logging</td>
</tr>
<tr>
<td>show-log</td>
<td>Show log entries</td>
</tr>
<tr>
<td>clear-log</td>
<td>Clear pending log entries</td>
</tr>
<tr>
<td>list-rules</td>
<td>List all defined Rules</td>
</tr>
<tr>
<td>add-rule</td>
<td>Define a new Rule</td>
</tr>
<tr>
<td>set-rule</td>
<td>Configure a Rule</td>
</tr>
<tr>
<td>remove-rule</td>
<td>Delete a Rule</td>
</tr>
<tr>
<td>remove-all-rules</td>
<td>Delete all Rules</td>
</tr>
<tr>
<td>enable-rule</td>
<td>Enable a Rule</td>
</tr>
<tr>
<td>disable-rule</td>
<td>Disable a Rule</td>
</tr>
<tr>
<td>disable-all-rules</td>
<td>Disable all Rules</td>
</tr>
<tr>
<td>rule-info</td>
<td>Display information about a Rule</td>
</tr>
<tr>
<td>reset-rule-stats</td>
<td>Reset all statistics of a Rule</td>
</tr>
<tr>
<td>list-alarms</td>
<td>List all defined Alarms</td>
</tr>
<tr>
<td>add-alarm</td>
<td>Define a new Alarm</td>
</tr>
<tr>
<td>set-alarm</td>
<td>Configure an Alarm</td>
</tr>
<tr>
<td>remove-alarm</td>
<td>Delete an Alarm</td>
</tr>
<tr>
<td>remove-all-alarms</td>
<td>Delete all Alarms</td>
</tr>
<tr>
<td>enable-alarm</td>
<td>Enable an Alarm</td>
</tr>
<tr>
<td>disable-alarm</td>
<td>Disable an Alarm</td>
</tr>
<tr>
<td>disable-all-alarms</td>
<td>Disable all Alarms</td>
</tr>
<tr>
<td>alarm-info</td>
<td>Display information about an Alarm</td>
</tr>
</tbody>
</table>

Each command has its own built-in help system as well, which can be invoked by selecting a
command and passing the `--help` option after the command:

```bash
> wrtd-tool set-alarm -h
usage: wrtd-tool set-alarm [-h] -d DELAY [-s SETUP] [-p PERIOD]
    [-c COUNT] [-e] <node_id> <alarm_id>
```

Positional arguments:

- `<node_id>`  The ID of the WRTD Node (int, can be hex with "0x" prefix)
- `<alarm_id>` The ID of the Alarm (string up to 15 characters)

Optional arguments:

- `-h`, `--help` show this help message and exit
- `-d DELAY`, `--delay DELAY` Set the delay for this Alarm wrt now. Default
unit is ns, but an 'n','u','m' or 's' can be appended to the value to set it explicitly to nano, micro, milli or full seconds

-s SETUP, --setup SETUP
Set the setup time for this Alarm. Default unit is ns, but an 'n','u','m' or 's' can be appended to the value to set it explicitly to nano, micro, milli or full seconds

-p PERIOD, --period PERIOD
Set the period for this Alarm. Default unit is ns, but an 'n','u','m' or 's' can be appended to the value to set it explicitly to nano, micro, milli or full

-c COUNT, --count COUNT
Set the repeat count for this Alarm

-e, --enable
Also enable the Alarm after configuring it.

If a command returns an Error Code, the underlying Python Wrapper will raise an OSErrror exception and will provide all the available details:

> wrtd-tool remove-alarm 1 alarm5
OSErrror: [Errno -1074122744] WRTD_ERROR_ALARM_DOES_NOT_EXIST
    wrtd_remove_alarm/wrtd_find_alarm: The specified alarm has not___
    been defined

Here’s an example on how to configure a Rule and check the Event Log for Events:

> wrtd-tool list-nodes
-> WRTD Node detected with ID: 10
> wrtd-tool list-rules 10
0 Rules defined.
> wrtd-tool add-rule 10 rule0
> wrtd-tool set-rule 10 rule0 LC-I1 NET0
> wrtd-tool list-rules 10 -v
1 Rule defined:
+ rule0
  + Configuration
    - Name...............: rule0
    - Source............: LC-I1
    - Destination.......: NET0
    - Enabled...........: False
    - Send Late.........: True
    - Repeat Count......: 0
    - Delay.............: 0.000ns
    - Holdoff...........: 0.000ns
    - Resync Period.....: 0.000ns
    - Resync Factor.....: 0
  + Statistics
    - RX Events.........: 0
    - Last RX...........: Never

(continues on next page)
- TX Events.........: 0
- Last TX...........: Never
- Latency (min).....: 0.000ns
- Latency (avg).....: 0.000ns
- Latency (max).....: 0.000ns
- Missed (late).....: 0
- Missed (holdoff)...: 0
- Missed (no sync)...: 0
- Missed (overflow): 0
- Last Missed.......: Never

```bash
> wrtd-tool enable-rule 10 rule0
> wrtd-tool enable-log 10
> wrtd-tool show-log 10 -c 6

<table>
<thead>
<tr>
<th>Id</th>
<th>Seq</th>
<th>...</th>
<th>...</th>
<th>...</th>
<th>...</th>
<th>GENERATED</th>
<th>DEVICE_0</th>
</tr>
</thead>
<tbody>
<tr>
<td>LC-I1</td>
<td>0000</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>NETWORK</td>
<td>TX</td>
</tr>
<tr>
<td>NET0</td>
<td>0016</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>NETWORK</td>
<td>TX</td>
</tr>
<tr>
<td>LC-I1</td>
<td>0000</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>GENERATE</td>
<td>DEVICE_0</td>
</tr>
<tr>
<td>NET0</td>
<td>0017</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>TX</td>
<td></td>
</tr>
<tr>
<td>LC-I1</td>
<td>0000</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>GENERATE</td>
<td>DEVICE_0</td>
</tr>
<tr>
<td>NET0</td>
<td>0018</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>TX</td>
<td></td>
</tr>
</tbody>
</table>
```
CHAPTER 5

Reference Nodes

WRTD provides so-called “reference” Nodes for the most common use-cases. Currently these include:

1. **SPEC150T-based FMC-ADC**: A four channel, 100MHz, 14bit PCIe-based ADC capable of triggering via WRTD.

2. **SVEC-based TDC+FD**: A pulse-in/pulse-out WRTD Node in VME format, for generic trigger distribution applications.

### 5.1 SPEC150T-based FMC-ADC

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td># of Applications</td>
<td>1</td>
</tr>
<tr>
<td>Name</td>
<td>wrtd-adc</td>
</tr>
<tr>
<td>Local input Channels</td>
<td>5</td>
</tr>
<tr>
<td>Local output Channels</td>
<td>1</td>
</tr>
<tr>
<td>Maximum Rules</td>
<td>16</td>
</tr>
<tr>
<td>Maximum Alarms</td>
<td>1</td>
</tr>
<tr>
<td>Average input to Message latency</td>
<td>$12\mu s$</td>
</tr>
<tr>
<td>Average Message to output latency</td>
<td>$12\mu s$</td>
</tr>
<tr>
<td>Can receive Messages over WR</td>
<td>YES</td>
</tr>
<tr>
<td>Can send Messages over WR</td>
<td>YES</td>
</tr>
</tbody>
</table>

This is a WRTD Node based on the Simple PCIe FMC Carrier (SPEC) and the FMC ADC 100M 14b 4cha (FMC-ADC).

**Important:** This Node does not use the standard SPEC, because the FPGA (XC6SLX45T) is not large enough for the complete design. Instead it uses the pin-compatible XC6SLX150T FPGA.
This Node provides the possibility to generate WRTD Messages based on trigger events of the FMC-ADC, as well as to trigger the FMC-ADC from incoming WRTD Messages.

The architecture of the Node can be seen in Fig. 5.1. The user communicates with the Node using one of the methods mentioned in Section 4. At the same time, the user accesses the FMC-ADC core to configure everything not related to WRTD (acquisition parameters, trigger source selection, data retrieval, etc.) using the existing ADC library. Internally, the Node is running one Application, responsible for configuring the WRTD-related aspects of the FMC-ADC.

The five Local Input Channels are mapped as follows:

<table>
<thead>
<tr>
<th>Channel</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>LC-I1</td>
<td>ADC Channel #1 internal trigger</td>
</tr>
<tr>
<td>LC-I2</td>
<td>ADC Channel #2 internal trigger</td>
</tr>
<tr>
<td>LC-I3</td>
<td>ADC Channel #3 internal trigger</td>
</tr>
<tr>
<td>LC-I4</td>
<td>ADC Channel #4 internal trigger</td>
</tr>
<tr>
<td>LC-I5</td>
<td>External trigger input</td>
</tr>
</tbody>
</table>

Note: In order for any of the Local Input Channels to produce an Event, the user must also properly configure the trigger source of the FMC-ADC via the ADC library. This falls outside the scope of WRTD library.

The single Local Output Channel (LC-O1) is mapped to the “WRTD” trigger input of the FMC-ADC.

Note: In order for Local Output Channel to trigger the FMC-ADC, the user must also properly
configure the trigger source of the FMC-ADC via the ADC library, to enable triggering from the “WRTD” trigger input. This falls outside the scope of WRTD library.

**Hint:** The User Application and the WRTD Application access different and separate parts of the FMC-ADC, so there is no danger of accessing the same resources simultaneously.

### 5.2 SVEC-based TDC+FD

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td># of Applications</td>
<td>2</td>
</tr>
<tr>
<td>Name</td>
<td></td>
</tr>
<tr>
<td>App #1</td>
<td>wrtd-tdc</td>
</tr>
<tr>
<td>App #2</td>
<td>wrtd-fd</td>
</tr>
<tr>
<td>Local input Channels</td>
<td>5</td>
</tr>
<tr>
<td>Local output Channels</td>
<td>0</td>
</tr>
<tr>
<td>Maximum Rules</td>
<td>16</td>
</tr>
<tr>
<td>Maximum Alarms</td>
<td>0</td>
</tr>
<tr>
<td>Average input to Message latency</td>
<td>20µs</td>
</tr>
<tr>
<td>Average Message to output latency</td>
<td>N.A.</td>
</tr>
<tr>
<td>Can receive Messages over WR</td>
<td>NO</td>
</tr>
<tr>
<td>Can send Messages over WR</td>
<td>YES</td>
</tr>
</tbody>
</table>

This is a WRTD *Node* based on the Simple VME FMC Carrier (SVEC), the FMC Time to Digital Converter (FMC-TDC) and the FMC Fine Delay generator (FMC-FD).

**Important:** The FMC-TDC should always be attached to “FMC Slot 1” of the SVEC, and the FMC-FD should always be attached to “FMC Slot 2”. It is not necessary though to have both the FMCs attached, the *Node* will work if only one of the two (TDC or FD) is present, as long as it is connected to the correct FMC slot.

The basic principle of this *Node* is simple: it takes in external pulses on its FMC-TDC inputs, timestamps them using WR time and converts them to WRTD *Messages*, to be sent over the WR network. Conversely, the *Node* also receives WRTD *Messages* which are then used to generate pulses at a predefined moment on one of the FMC-FD outputs. As such, it can be seen as a “pulse-to-message” and “message-to-pulse” converter with applications in the fields of pulse distribution, trigger synchronisation, etc.

The architecture of the *Node* can be seen in Fig. 5.2. The user communicates with the *Node* via the WRTD library, using one of the methods mentioned in Section 4. Internally, the *Node* is running two *Applications*, each on a dedicated CPU. One is responsible for the FMC-TDC peripheral, while the other handles the FMC-FD.

The five *Local Input Channels* are mapped to the five inputs of the FMC-TDC. Thus, *Event ID LC-I1* corresponds to the first FMC-TDC channel, *LC-I2* to the second one, and so on, up to *LC-I5*. 

5.2. SVEC-based TDC+FD
Fig. 5.2: SVEC-based TDC+FD reference WRTD Node
The four *Local Output Channels* are mapped to the four outputs of the FMC-FD. *Event ID LC-O1* corresponds to the first FMC-FD output, and so on, up to *LC-O4*.

**Hint:** It is possible for the User Application in Fig. 5.2 to access directly the FMC-TDC and FMC-FD cores by means of their respective libraries (shown in Fig. 5.2 with dotted lines), in order to fine-tune them and/or change their default behaviour. However this should be done with extreme care, as it could lead to a race condition between the User Application and the WRTD Applications running inside the FPGA.

The front-panel LEMO connectors of the SVEC are used as follows:

<table>
<thead>
<tr>
<th>Connector</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>WR 1-PPS output</td>
</tr>
<tr>
<td>L2</td>
<td>Not used</td>
</tr>
<tr>
<td>L3</td>
<td>WR external reference clock input</td>
</tr>
<tr>
<td>L4</td>
<td>WR external 1-PPS input</td>
</tr>
</tbody>
</table>

The front-panel LED indicators have the following meanings:

<table>
<thead>
<tr>
<th>Function</th>
<th>LED</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>WR link activity</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>FMC-FD clock locked to WR</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>WR locked</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>VME bus access</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

**Hint:** LED colors follow a certain convention:

- A green LED means *OK* or *on*.
- A LED that is turned off means *off* or *not active*.
- A red LED means *error*.
- An orange LED signifies *activity* (used by LEDs 1, 4 and 5).
This section provides guidelines on how to develop a new WRTD Node (or, modify an existing one), including Hardware, Gateware and Firmware, using the resources provided by the WRTD project. If you simply need to control an existing WRTD Node, please refer to Section 4 instead.

It should be noted that a new Node does not necessarily need to use the resources provided by the WRTD project itself. In fact, the only hard requirement is that the Node must be able to send and/or receive Event Messages over an Ethernet-based LAN, using multicast UDP on address 224.0.23.159, port 5044. It should also respect the Event ID naming conventions of WRTD.

Of course, the above is the absolute minimum requirement, which is already fullfilled by any LXI device supporting the relevant LXI extended functions (see also Relation to IVI and LXI).

Moving one step further, the network interface of the Node should be White Rabbit enabled, to allow for accurate timestamping and sub-ns synchronisation between the Nodes. This has implications for your Hardware.

If your Node contains an FPGA and you choose to use the WRTD-provided Gateware resources, then this also opens up the possibility of using the WRTD-provided Firmware resources as well, to develop the application(s) running inside the FPGA. A WRTD Node developed with the WRTD Gateware and Firmware resources will be accessible via the provided C Library, Python Wrapper and Tools.

### 6.1 Setting Up the Environment

The WRTD development environment has been tested and works best under Linux.

The source code of WRTD is maintained in a Git repository. To create a local copy of it, please use:

```
$ git clone "https://ohwr.org/project/wrtd.git"
```
And then checkout the correct commit, branch or tag that you wish to work on. Typically, the latest stable commit will be tracked by the master branch, while the latest development commit will be tracked by the proposed_master branch.

Once this is done, you need to bring in the various dependencies of WRTD (which are present as Git submodules within the WRTD repository):

```
$ cd wrtd
$ git submodule update --init
```

**Important:** Despite the fact that many of WRTD's dependencies contain their own submodules, you should not bring in those dependencies recursively, as they may create conflicts (e.g. both WRTD and one of its submodules depending on another submodule but on a different version of it). WRTD's top-level dependencies (which are brought in with the above command) contain everything that is needed for development.

Furthermore, you will need a RISC-V cross-compilation toolchain for compiling the firmware for the Nodes.

**Note:** Because at the time of the release of WRTD v1.0, the OHWR deployment procedures were undergoing significant changes (in particular with respect to packaging), up-to-date installation instructions for all the necessary development tools will be available through the project Wiki. Once these procedures have been finalised and tested, the contents of the wiki page will be merged here.

Alternatively, you can try to build your own toolchain like this:

```
$ git clone "https://ohwr.org/project/soft-cpu-toolchains.git"
$ cd soft-cpu-toolchains/riscv
$ sh ./build-riscv.sh
```

This will checkout the master branch of the repository and start the build process, which takes a long time to complete.

Once the process is complete, the resulting RISC-V toolchain binaries will be placed under soft-cpu-toolchains/riscv/riscv-toolchain/bin. You should either copy them to somewhere within your system’s path (typically /usr/local/bin), or set the environment variable CROSS_COMPILE_TARGET to point to the correct location, with the riscv32-elf- suffix appended to the path, like this:

```
$ export CROSS_COMPILE_TARGET="<path_to_repository>/riscv/riscv-toolchain/bin/riscv32-elf-
```

You either need to configure the above command to be run automatically on a new terminal (typically by putting the command in ~/.profile or ~/.bash_profile but this depends ultimately on the shell you are using), or you should run it every time you start a new terminal and need to compile RISC-V firmware.
6.2 Hardware

Hardware-wise, WRTD does not impose any particular requirements. It is assumed that the device is capable of producing and/or consuming “events” (e.g. trigger in, trigger out), otherwise there is no point in using WRTD.

Other than that, the only requirement on the hardware is that it supports White Rabbit (WR).

If you are designing your own board and you would like to have something as a reference, here’s a page which can help you, or, more specifically, here’s a list with a few open-source boards that already support WR:

- SPEC (Xilinx Spartan6 FPGA)
- FASEC (Xilinx Zynq SoC FPGA)
- VFC-HD (Intel Arria V)

All of the above examples make use of the WR PTP Core inside their FPGA.

6.3 Gateware

There is no HDL template yet for WRTD gateware development. Users who need to develop WRTD gateware for a new Node are advised to use the top-level VHDL modules of the Reference Nodes as a starting point. Their source code is available on the WRTD Git repository:

https://www.ohwr.org/project/wrtd/tree/master/hdl/top

Every Node should include at a minimum an instance of MockTurtle and the WR PTP Core.

Furthermore, the MockTurtle configuration should define at least one Host Message Queue per CPU, for communication between the WRTD library and the WRTD Application running on the CPU, as well as one Remote Message Queue per CPU for connecting the WRTD Application to the WR network.

Every Node should also include a MockTurtle Ethernet Endpoint, to interface the Remote Message Queue of the MockTurtle with the Fabric Interface of the WR PTP Core. If in doubt, have a look at how these three modules are instantiated and connected to each other in the Reference Nodes.

6.4 Firmware

WRTD provides a common firmware development framework for Applications. To use it, users must also use a compatible Gateware.

Similar to Gateware development, users are advised to use the firmware of the Reference Nodes as a starting point:

https://www.ohwr.org/project/wrtd/tree/master/software/firmware

6.4.1 Setup

To start developing firmware for WRTD, users must first include the relevant header file:
Users must also describe the various features of their application, by means of C preprocessor directives. The following needs to be copied to the application source code and adjusted as necessary:

```c
#include "wrdt-rt-common.h"

/* Number of MockTurtle CPUs. */
#define NBR_CPUS xxx
/* Index of this CPU. */
#define CPU_IDX xxx
/* Maximum number of Rules allowed */
#define NBR_RULES xxx
/* Maximum number of Alarms allowed */
#define NBR_ALARMS xxx
/* Number of Devices. A "Device" is a unidirectional set of Local Channels.
Most Applications will have one Device per direction (in/out).
Maximum allowed number of Devices = 4. */
#define NBR_DEVICES xxx
/* Number of Local Channels per Device. */
#define DEVICES_NBR_CHS { xxx, 0, 0, 0}
/* Direction of Local Channels per Device. Direction can be either
wrtd_ch_dir_in (from the environment to wrtd) or
wrtd_ch_dir_out (from wrtd to the environment). */
#define DEVICES_CHS_DIR { WRTD_CH_DIR_IN, 0, 0, 0}
/* A unique number to identify this Application. */
#define APP_ID xxx
/* Application version (major, minor). */
#define APP_VER RT_VERSION(xxx, yyy)
/* A string name for this Application. */
#define APP_NAME xxx
/* 1 if the Application can receive Events over the network,
0 otherwise. */
#define WRTD_NET_RX x
/* 1 if the Application can send Events over the network,
0 otherwise. */
#define WRTD_NET_TX x
/* 1 if the Application can receive Events from Local Channels,
0 otherwise. */
#define WRTD_LOCAL_RX x
/* 1 if the Application can send Events to Local Channels,
0 otherwise. */
#define WRTD_LOCAL_TX x
```

Apart from the preprocessor definitions, users must also provide implementations for a set of functions. These include:

- **Initialisation and Main Loop**
  - wrtd_user_init()
  - wrtd_io()
• **WR Link Control and Status**
  - `wr_link_up()`
  - `wr_time_ready()`
  - `wr_enable_lock()`
  - `wr_time_locked()`
  - `wr_sync_timeout()`

• **Sending Events**
  - `wrtd_local_output()`

### 6.4.2 Initialisation and Main Loop

WRTD firmware runs in a continuous loop. All such code should be put in the `wrtd_io()` function. Additionally, the `wrtd_user_init()` offers the possibility to execute code once, after reset, before entering the main execution loop.

```c
static int wrtd_user_init (void)
{
    Function to perform any application-specific initialisation (runs once after reset).
    If the application does not need this, write a function that returns always 0.

    **Return** 0 on success, or error code otherwise.
}
```

```c
static void wrtd_io (void)
{
    Function to perform the main tasks of the application. This will run in a loop.

    Typically the application will check here for incoming Events and pass them to WRTD via the `wrtd_route_in()` function. It might also want to check if output Events previously scheduled via `wrtd_local_output()` have been executed or not.

    If the application does not need this, write a function that returns always 0.

    **Return** 0 on success, or error code otherwise.
}
```

### 6.4.3 WR Link Control and Status

It is expected (but not required) that WRTD firmware has access to the WR link. In particular, most applications should be able to detect whether the link is up, the timecode is valid, etc. The way this is done is application-specific. The following functions allow users to describe how this is done.

```c
static int wr_link_up (void)
{
    Check whether the WR link is up or not.

    If the application does not need this, write a function that returns always 1.

    **Return** 1 if true, 0 otherwise.
}
static int wr_time_ready(void)
    Check whether the WR time is valid.
    If the application does not need this, write a function that returns always 1.

    Return 1 if true, 0 otherwise.

static void wr_enable_lock(int enable)
    Enable/disable locking of the application to the WR clock.
    If the application does not need this, write a function that does nothing.

    Parameters
    • enable: set to 1 to enable, set to 0 to disable.

static int wr_time_locked(void)
    Check whether the application is locked to the WR clock or not.
    If the application does not need this, write a function that returns always 1.

    Return 1 if true, 0 otherwise.

static int wr_sync_timeout(void)
    Return the delay in seconds that the application should wait before considering the WR
    link synced.
    If the application does not need this, write a function that returns always 0.

    Return Integer number of seconds to wait.

6.4.4 Event I/O

Ultimately, the purpose of a WRTD firmware is to relay Events from/to the outside world.
Internally, WRTD represents an Event using the wrtd_event structure.

struct wrtd_event
    WRTD Event

    Public Members

struct wrtd_tstamp ts
    Time of the event.

char id[WRTD_ID_LEN]
    Event id.

uint32_t seq
    Sequence number.

unsigned char flags
    Associated flags.
**Sending Events**

When an incoming *Event* has been matched to a *Rule* with a *Local Channel* output, WRTD will call the user-provided `wrtd_local_output()` function. This function should perform all the application-specific actions to program the relevant *Local Channel* to generate the actual output.

```c
static int wrtd_local_output(struct wrtd_event *ev, unsigned ch)
    Generate an output Event on a Local Channel.
    If the application does not need this, write a function that returns always 0.
    Return 0 on success, or error code otherwise.

Parameters
    • ev: pointer to a struct `wrtd_event`, representing the Event to send.
    • ch: Local Channel number to use.
```

**Receiving Events**

Receiving an *Event* is also application-specific. The monitoring of *Local Channel* inputs is typically done periodically in the main execution loop, using the `wrtd_io()` function.

Once the firmware detects an incoming *Event* and fills in a `wrtd_event` structure, it should simply pass this to WRTD using the `wrtd_route_in()` function. The rest (rule matching, event forwarding, etc.) will be handled by WRTD.

```c
static void wrtd_route_in(struct wrtd_event *ev)
    Route an event coming from a Local Input, Network, or Alarm.

Parameters
    • ev: pointer to a struct `wrtd_event`, representing the received Event.
```

**Hint:** Contrary to the rest of the functions presented in Section 6.4, `wrtd_route_in()` is not a user-defined function. It is already provided by the WRTD firmware development framework. Users should simply create and fill in the `wrtd_event ev` structure with the details of the received *Event* and then pass it on to WRTD by calling this function.
p
PyWrtd, 47
Symbols
__WRTD_ATTR_BASE (C++ enumerator), 24
__WRTD_ATTR_MAX_NUMBER (C++ enumerator), 26
__WRTD_ERROR_BASE (C++ enumerator), 18
__WRTD_ERROR_MAX_NUMBER (C++ enumerator), 19
__WRTD_LXISYNC_ERROR_BASE (C++ enumerator), 18
__WRTD_SPECIFIC_ERROR_BASE (C++ enumerator), 18

A
add_alarm() (PyWrtd.PyWrtd method), 44
add_rule() (PyWrtd.PyWrtd method), 45

C
clear_event_log_entries()
(PyWrtd.PyWrtd method), 43

disable_all_alarms() (PyWrtd.PyWrtd method), 44
disable_all_rules() (PyWrtd.PyWrtd method), 45

G
get_alarm_name() (PyWrtd.PyWrtd method), 45
get_attr_bool() (PyWrtd.PyWrtd method), 41
get_attr_int32() (PyWrtd.PyWrtd method), 42
get_attr_string() (PyWrtd.PyWrtd method), 42
get_attr_tstamp() (PyWrtd.PyWrtd method), 43
get_fw_name() (PyWrtd.PyWrtd method), 47

get_next_event_log_entry()
(PyWrtd.PyWrtd method), 43
get_node_count() (PyWrtd.PyWrtd static method), 41
get_node_id() (PyWrtd.PyWrtd static method), 41
get_rule_name() (PyWrtd.PyWrtd method), 46

P
PyWrtd (class in PyWrtd), 40
PyWrtd (module), 40, 47

R
remove_alarm() (PyWrtd.PyWrtd method), 44
remove_all_alarms() (PyWrtd.PyWrtd method), 44
remove_all_rules() (PyWrtd.PyWrtd method), 46
remove_rule() (PyWrtd.PyWrtd method), 46
reset() (PyWrtd.PyWrtd method), 41
reset_rule_stats() (PyWrtd.PyWrtd method), 46

S
set_attr_bool() (PyWrtd.PyWrtd method), 41
set_attr_int32() (PyWrtd.PyWrtd method), 42
set_attr_string() (PyWrtd.PyWrtd method), 42
set_attr_tstamp() (PyWrtd.PyWrtd method), 43

W
wr_enable_lock (C++ function), 62
wr_link_up (C++ function), 61
wr_sync_timeout (C++ function), 62
wr_time_locked (C++ function), 62
wr_time_ready (C++ function), 61
wrtd_add_alarm (C++ function), 33
wrtd_add_rule (C++ function), 35
wrtd_attr (C++ type), 23
WRTD_ATTR_ALARM_COUNT (C++ enumerator), 24
WRTD_ATTR_ALARM_ENABLED (C++ enumerator), 24
WRTD_ATTR_ALARM_PERIOD (C++ enumerator), 24
WRTD_ATTR_ALARM_REPEAT_COUNT (C++ enumerator), 24
WRTD_ATTR_ALARM_SETUP_TIME (C++ enumerator), 24
WRTD_ATTR_ALARM_TIME (C++ enumerator), 24
WRTD_ATTR_EVENT_LOG_EMPTY (C++ enumerator), 24
WRTD_ATTR_EVENT_LOG_ENABLED (C++ enumerator), 24
WRTD_ATTR_FW_CAPABILITIES (C++ enumerator), 26
WRTD_ATTR_FW_COUNT (C++ enumerator), 26
WRTD_ATTR_FW_LOCAL_INPUTS (C++ enumerator), 26
WRTD_ATTR_FW_LOCAL_OUTPUTS (C++ enumerator), 26
WRTD_ATTR_FW_MAJOR_VERSION (C++ enumerator), 26
WRTD_ATTR_FW_MAJOR_VERSION_REQUIRED (C++ enumerator), 26
WRTD_ATTR_FW_MAX_ALARMS (C++ enumerator), 26
WRTD_ATTR_FW_MAX_RULES (C++ enumerator), 26
WRTD_ATTR_FW_MINOR_VERSION (C++ enumerator), 26
WRTD_ATTR_FW_MINOR_VERSION_REQUIRED (C++ enumerator), 26
WRTD_ATTR_IS_TIME_SYNCHRONIZED (C++ enumerator), 24
WRTD_ATTR_RULE_COUNT (C++ enumerator), 24
WRTD_ATTR_RULE_DELAY (C++ enumerator), 25
WRTD_ATTR_RULE_DESTINATION (C++ enumerator), 24
WRTD_ATTR_RULE_ENABLED (C++ enumerator), 24
WRTD_ATTR_RULE_HOLDOFF (C++ enumerator), 25
WRTD_ATTR_RULE_REPEAT_COUNT (C++ enumerator), 24
WRTD_ATTR_RULE_RESYNC_FACTOR (C++ enumerator), 25
WRTD_ATTR_RULE_RESYNC_PERIOD (C++ enumerator), 25
WRTD_ATTR_RULE_SEND_LATE (C++ enumerator), 25
WRTD_ATTR_RULE_SOURCE (C++ enumerator), 24
WRTD_ATTR_STAT_RULE_MISSED_HOLDOFF (C++ enumerator), 25
WRTD_ATTR_STAT_RULE_MISSED_LAST (C++ enumerator), 25
WRTD_ATTR_STAT_RULE_MISSED_LATE (C++ enumerator), 25
WRTD_ATTR_STAT_RULE_MISSED_NOSYNC (C++ enumerator), 25
WRTD_ATTR_STAT_RULE_MISSED_OVERFLOW (C++ enumerator), 25
WRTD_ATTR_STAT_RULE_RX_EVENTS (C++ enumerator), 25
WRTD_ATTR_STAT_RULE_RX_LAST (C++ enumerator), 25
WRTD_ATTR_STAT_RULE_RX_LATENCY_AVG (C++ enumerator), 25
WRTD_ATTR_STAT_RULE_RX_LATENCY_MAX (C++ enumerator), 25
WRTD_ATTR_STAT_RULE_RX_LATENCY_MIN (C++ enumerator), 25
WRTD_ATTR_STAT_RULE_RX_RESERVED5 (C++ enumerator), 25
WRTD_ATTR_STAT_RULE_RX_RESERVED6 (C++ enumerator), 25
WRTD_ATTR_STAT_RULE_RX_RESERVED7 (C++ enumerator), 25
wrt_close (C++ function), 21
wrt_disable_all_alarms (C++ function), 34
wrt_disable_all_rules (C++ function), 36
WRTD_ERROR_ALARM_DOES_NOT_EXIST (C++ enumerator), 18
WRTD_ERROR_ALARM_EXISTS (C++ enumerator), 18
WRTD_ERROR_ATTR_GLOBAL (C++ enumerator), 19
WRTD_ERROR_ATTR_NOT_READABLE (C++ enumerator), 18

Index
WRTD_ERROR_ATTR_NOT_WRITEABLE (C++ enumerator), 18
WRTD_ERROR_BADLY_FORMED_SELECTOR (C++ enumerator), 18
WRTD_ERROR_INTERNAL (C++ enumerator), 19
WRTD_ERROR_INVALID_ATTRIBUTE (C++ enumerator), 18
WRTD_ERROR_INVALID_VALUE (C++ enumerator), 18
wrtd_error_message (C++ function), 19
WRTD_ERROR_NOT_INITIALIZED (C++ enumerator), 18
WRTD_ERROR_NULL_POINTER (C++ enumerator), 18
WRTD_ERROR_OUT_OF_MEMORY (C++ enumerator), 18
WRTD_ERROR_OUT_OF_RESOURCES (C++ enumerator), 19
WRTD_ERROR_RESOURCE_ACTIVE (C++ enumerator), 19
WRTD_ERROR_RESOURCE_UNKNOWN (C++ enumerator), 18
WRTD_ERROR_RULE_DOES_NOT_EXIST (C++ enumerator), 19
WRTD_ERROR_RULE_EXISTS (C++ enumerator), 19
WRTD_ERROR_UNEXPECTED_RESPONSE (C++ enumerator), 18
WRTD_ERROR_UNKNOWN_CHANNEL_NAME (C++ enumerator), 18
WRTD_ERROR_UNKNOWN_LOG_TYPE (C++ enumerator), 19
WRTD_ERROR_VERSION_MISMATCH (C++ enumerator), 18
wrtd_event (C++ class), 62
wrtd_event::flags (C++ member), 62
wrtd_event::id (C++ member), 62
wrtd_event::seq (C++ member), 62
wrtd_event::ts (C++ member), 62
wrtd_get_alarm_name (C++ function), 34
wrtd_get_attr_bool (C++ function), 27
wrtd_get_attr_int32 (C++ function), 28
wrtd_get_attr_string (C++ function), 28
wrtd_get_attr_tstamp (C++ function), 29
wrtd_get_error (C++ function), 19
wrtd_get_fw_name (C++ function), 39
wrtd_get_next_event_log_entry (C++ function), 31
wrtd_get_node_count (C++ function), 22
wrtd_get_node_id (C++ function), 22
wrtd_get_rule_name (C++ function), 36
WRTD_GLOBAL_REP_CAP_ID (C macro), 23
wrtd_init (C++ function), 21
wrtd_io (C++ function), 61
wrtd_local_output (C++ function), 63
WRTD_LOG_ENTRY_SIZE (C macro), 33
wrtd_remove_alarm (C++ function), 34
wrtd_remove_all_alarms (C++ function), 34
wrtd_remove_all_rules (C++ function), 36
wrtd_remove_rule (C++ function), 36
wrtd_reset (C++ function), 21
wrtd_reset_rule_stats (C++ function), 37
wrtd_route_in (C++ function), 63
wrtd_set_attr_bool (C++ function), 27
wrtd_set_attr_int32 (C++ function), 27
wrtd_set_attr_string (C++ function), 28
wrtd_set_attr_tstamp (C++ function), 29
wrtd_status (C++ type), 18
WRTD_SUCCESS (C++ enumerator), 18
wrtd_tstamp (C++ class), 30
wrtd_tstamp::frac (C++ member), 30
wrtd_tstamp::ns (C++ member), 30
wrtd_tstamp::seconds (C++ member), 30
wrtd_user_init (C++ function), 61

Index 69