



Xena OpenAutomation Core Documentation

Release 2.0.3

Xena Networks

Jan 03, 2024

TABLE OF CONTENT

1	Introduction	3
2	Getting Started	5
3	Understanding XOA Core	17
4	API Reference	27
5	Glossary of Terms	29
6	Indices and Tables	31
	Index	33

Xena OpenAutomation Core (*XOA Core*) is a framework for executing Xena automated test suites and managing Xena's physical and virtual Traffic Generation and Analysis (*TGA*) testers.

The *Xena OpenAutomation Core Documentation* describes the basic architecture and how to run Xena's automated test suites.

The target audience of this document is test specialists who develop and run automated test scripts/programs using Xena *TGA* hardware and software. Users of this document should have the following knowledge and experience:

- Ability to program with Python language.
- Familiarity with the operating system of your development environment.
- Familiarity with Xena test equipment.
- Working knowledge of data communications theory and practice.

Important: To learn *XOA Python API*, go to [Xena OpenAutomation Python API Documentation](#).

To learn *XOA CLI* commands, go to [Xena OpenAutomation CLI Command Documentation](#).

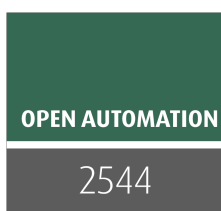
INTRODUCTION

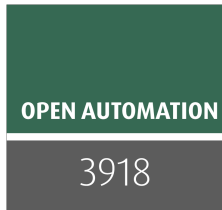
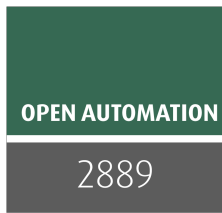
Xena OpenAutomation Core (*XOA Core*) is an open-source test suite framework for network automation and testing. It is designed to host various *XOA Test Suites* as plugins, allowing users to create, manage, and run test cases for different network scenarios. The XOA Core framework serves as the foundation for building and executing test suites in the XOA ecosystem.

Key features of XOA Core include:

1. **Modular architecture:** The test suite framework employs a modular architecture, enabling users to develop and run different test suites as plugins.
2. **Test Suite execution:** XOA Core supports both local and remote test suite execution. Users can execute test suites on their local machines or on remote testbeds through the XOA CLI or Web GUI.
3. **Test Case management:** XOA Core provides tools for managing test cases, including creating, updating, and deleting them. Users can also organize test cases using tags and execute them in parallel or sequentially.
4. **Extensibility:** The framework is designed to be extensible, allowing users to develop custom test suites and plugins to address specific testing requirements.
5. **Logging and reporting:** XOA Core offers built-in logging and reporting functionality, generating detailed test reports to help users analyze test results and identify issues.

The [official documentation](#) provides a comprehensive guide on how to install, configure, and use XOA Core. It covers topics such as setting up the environment, creating test suites, executing test cases, and analyzing test results. The documentation also includes a reference for XOA Core's API and example test suites to help users get started with their test automation projects.





GETTING STARTED

2.1 Installing XOA Core

XOA Core is available to install via the [Python Package Index](#). You can also install from the source file.

2.1.1 Prerequisites

Before installing XOA Core, please make sure your environment has installed:

- *Python*
- *pip*

Python

XOA Core requires that you [install Python](#) on your system.

Note: XOA Core requires Python ≥ 3.8 .

pip

Make sure `pip` is installed on your system. `pip` is the [package installer for Python](#). You can use it to install packages from the [Python Package Index](#) and other indexes.

Usually, `pip` is automatically installed if you are:

- working in a [virtual Python environment](#) (`virtualenv` or `venv`). It is not necessary to use `sudo pip` inside a virtual Python environment.
- using Python downloaded from [python.org](#)

If you don't have `pip` installed, you can:

- Download the script, from <https://bootstrap.pypa.io/get-pip.py>.

- Open a terminal/command prompt, cd to the folder containing the `get-pip.py` file and run:

Listing 1: Install pip in Windows environment.

```
> py get-pip.py
```

Listing 2: Install pip in macOS/Linux environment.

```
$ python3 get-pip.py
```

See also:

Read more details about this script in [pypa/get-pip](#).

Read more about installation of `pip` in [pip installation](#).

2.1.2 Installing From PyPI Using pip

`pip` is the recommended installer for XOA Core. The most common usage of `pip` is to install from the [Python Package Index](#) using [Requirement Specifiers](#).

Note: If you install XOA Core using `pip install xoa-core`, XOA Python API (PyPI package name `xoa_driver`) will be automatically installed.

Install to Global Namespace

Listing 3: Install XOA Core in Windows environment from PyPI.

```
> pip install xoa-core           # latest version
> pip install xoa-core==1.0.7   # specific version
> pip install xoa-core>=1.0.7   # minimum version
```

Listing 4: Install XOA Core in macOS/Linux environment from PyPI.

```
$ pip install xoa-core           # latest version
$ pip install xoa-core==1.0.7   # specific version
$ pip install xoa-core>=1.0.7   # minimum version
```

Install in Virtual Environment

Install XOA Core in a virtual environment, so it does not pollute your global namespace.

For example, your project folder is called `/my_xoa_project`.

Listing 5: Install XOA Core in a virtual environment in Windows from PyPI.

```
[my_xoa_project]> python -m venv ./env
[my_xoa_project]> source ./env/bin/activate

(env) [my_xoa_project]> pip install xoa-core           # latest version
(env) [my_xoa_project]> pip install xoa-core==1.0.7   # specific version
(env) [my_xoa_project]> pip install xoa-core>=1.0.7   # minimum version
```

Listing 6: Install XOA Core in a virtual environment in macOS/Linux from PyPI.

```
[my_xoa_project]$ python3 -m venv ./env
[my_xoa_project]$ source ./env/bin/activate

(env) [my_xoa_project]$ pip install xoa-core           # latest version
(env) [my_xoa_project]$ pip install xoa-core==1.0.7   # specific version
(env) [my_xoa_project]$ pip install xoa-core>=1.0.7   # minimum version
```

Afterwards, your project folder will be:

Listing 7: After creating Python virtual environment

```
/my_xoa_project
|
|- env
```

See also:

- [Virtual Python environment](#)
- [virtualenv](#)
- [venv](#)

2.1.3 Upgrading From PyPI Using pip

To upgrade XOA Core package from PyPI:

Listing 8: Upgrade XOA Core in Windows environment from PyPI.

```
> pip install xoa-core --upgrade
```

Listing 9: Upgrade XOA Core in macOS/Linux environment from PyPI.

```
$ pip install xoa-core --upgrade
```

Note: If you upgrade XOA Core using `pip install --upgrade xoa-core`, XOA Python API (PyPI package name `xoa_driver`) will be automatically upgraded.

2.1.4 Installing Manually From Source

If for some reason you need to install XOA Core manually from source, the steps are:

Step 1, make sure Python packages `wheel` and `setuptools` are installed on your system. Install `wheel` and `setuptools` using `pip`:

Listing 10: Install `wheel` and `setuptools` in Windows environment.

```
> pip install wheel setuptools
```

Listing 11: Install `wheel` and `setuptools` in macOS/Linux environment.

```
$ pip install wheel setuptools
```

Step 2, download the XOA Core source distribution from [XOA Core Releases](#). Unzip the archive and run the `setup.py` script to install the package:

Listing 12: Install XOA Core in Windows environment from source.

```
[xoa_core]> python setup.py install
```

Listing 13: Install XOA Core in macOS/Linux environment from source.

```
[xoa_core]$ python3 setup.py install
```

Step 3, if you want to distribute, you can build .whl file for distribution from the source:

Listing 14: Build XOA Core wheel in Windows environment for distribution.

```
[xoa_core]> python setup.py bdist_wheel
```

Listing 15: Build XOA Core wheel in macOS/Linux environment for distribution.

```
[xoa_core]$ python3 setup.py bdist_wheel
```

Important: If you install XOA Core from the source code, you need to install XOA Python API (PyPI package name [xoa_driver](#)) separately. This is because XOA Python API is treated as a 3rd-party dependency of XOA Core. You can go to [XOA Python API](#) repository to learn how to install it.

2.1.5 Uninstall and Remove Unused Dependencies

`pip uninstall xoa-core` can uninstall the package itself but not its dependencies. Leaving the package's dependencies in your environment can later create conflicting dependencies problem.

We recommend install and use the [pip-autoremove](#) utility to remove a package plus unused dependencies.

Listing 16: Uninstall XOA Core in Windows environment.

```
> pip install pip-autoremove
> pip-autoremove xoa-core -y
```

Listing 17: Uninstall XOA Core in macOS/Linux environment.

```
$ pip install pip-autoremove
$ pip-autoremove xoa-core -y
```

See also:

See the [pip uninstall](#) reference.

See [pip-autoremove](#) usage.

2.2 Step-by-Step Guide

This section provides a step-by-step guide on how to use XOA Core to run XOA test suites.

2.2.1 Create Project Folder

To run XOA test suites, you need a folder to place the test suite plugins, the test configuration files, and your Python script to control the tests.

Let's create a folder called `/my_xoa_project`

Listing 18: Create the project folder

```
/my_xoa_project
|
```

2.2.2 Install XOA Core

After creating the folder, you can either choose to *install XOA Core in a Python virtual environment* or *install in your global namespace*.

If you have already installed XOA Core in your system, either to your global namespace or in a virtual environment, you can skip this step.

2.2.3 Place Test Suite Plugins

Depending on what XOA test you want to run, place the corresponding XOA test suite plugins and the test configuration files in `/my_xoa_project`.

Your project folder will look like this afterwards.

Listing 19: Copy test suite plugins into the project folder

```
/my_xoa_project
|
|- /test_suites
|   |- /plugin2544
|   |- /plugin2889
|   |- /plugin3918
```

2.2.4 Run Tests from XOA Test Suite Configurations

Important: If you run **Valkyrie test suite configuration files** (.v2544 for *Valkyrie2544*, .v2889 for *Valkyrie2889*, .v3918 for *Valkyrie3918*, and .v1564 for *Valkyrie1564*), go to *Run Tests from Valkyrie Test Suite Configurations*.

Copy your XOA test configuration .json files into /my_xoa_project for easy access. Then create a main.py file inside the folder /my_xoa_project.

Listing 20: Copy XOA test configs and create main.py

```
/my_xoa_project
|
|- main.py
|- new_2544_config.json
|- new_2889_config.json
|- new_3918_config.json
|- /test_suites
|   |- /plugin2544
|   |- /plugin2889
|   |- /plugin3918
```

This main.py controls the test workflow, i.e. load the configuration files, start tests, receive test results, and stop tests. The example below demonstrates a basic flow for you to run XOA tests.

```
from __future__ import annotations
from xoa_core import (
    controller,
    types,
)
import asyncio
import json
from pathlib import Path

PROJECT_PATH = Path(__file__).parent
XOA_CONFIG = PROJECT_PATH / "xoa_2544_config.json"
```

(continues on next page)

(continued from previous page)

```

PLUGINS_PATH = PROJECT_PATH / "test_suites"

async def subscribe(ctrl: "controller.MainController", channel_name: str,
    fltr: set["types.EMsgType"] | None = None) -> None:
    async for msg in ctrl.listen_changes(channel_name, _filter=fltr):
        print(msg)

async def main() -> None:
    # Define your tester login credentials
    my_tester_credential = types.Credentials(
        product=types.EProductType.VALKYRIE,
        host="10.20.30.40"
    )

    # Create a default instance of the controller class.
    ctrl = await controller.MainController()

    # Register the plugins folder.
    ctrl.register_lib(str(PLUGINS_PATH))

    # Add tester credentials into teh controller. If already added, it
    # will be ignored.
    # If you want to add a list of testers, you need to iterate
    # through the list.
    await ctrl.add_tester(my_tester_credential)

    # Subscribe to test resource notifications.
    asyncio.create_task(subscribe(ctrl, channel_name=types.PIPE_
    RESOURCES))

    # Load your XOA 2544 config and run.
    with open(XOA_CONFIG, "r") as f:

        # Get rfc2544 test suite information from the core's
        # registration
        info = ctrl.get_test_suite_info("RFC-2544")
        if not info:
            print("Test suite RFC-2544 is not recognized.")
            return None

        # Test suite name: "RFC-2544" is received from call of c.get_
        # available_test_suites()
        test_exec_id = ctrl.start_test_suite("RFC-2544", json.load(f))

```

(continues on next page)

(continued from previous page)

```

# The example here only shows a print of test result data.
asyncio.create_task(
    subscribe(ctrl, channel_name=test_exec_id, fltr={types.
→EMsgType.STATISTICS})
)

# By the next line, we prevent the script from being immediately
# terminated as the test execution and subscription are non-
→blockable, and they ran asynchronously,
await asyncio.Event().wait()

if __name__ == "__main__":
    asyncio.run(main())

```

Then simply run main.py:

Listing 21: Run test suite in Windows.

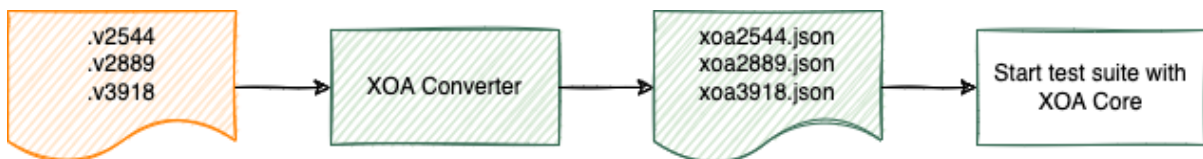
```
[my_xoa_project]> python main.py
```

Listing 22: Run test suite in macOS/Linux.

```
[my_xoa_project]$ python3 main.py
```

2.2.5 Run Tests from Valkyrie Test Suite Configurations

If you want to run your Valkyrie test suite configuration files, you should install `xoa-converter` to convert Valkyrie test suite configurations into XOA test suite configurations, as illustrated below.



See also:

Read more about installing [XOA Config Convert](#)

Copy your Valkyrie test configurations into `/my_xoa_project` for easy access. Then create a `main.py` file inside the folder `/my_xoa_project`.

Listing 23: Copy Valkyrie test configs and create main.py

```

/my_xoa_project
|
|- main.py

```

(continues on next page)

(continued from previous page)

```
| - old_2544_config.v2544
| - old_2889_config.v2889
| - old_3918_config.v3918
| - /test_suites
|   | - /plugin2544
|   | - /plugin2889
|   | - /plugin3918
```

This `main.py` controls the test workflow, i.e. convert Valkyrie configs into XOA configs, load the configuration files, start tests, receive test results, and stop tests. The example below demonstrates a basic flow for you to run Valkyrie tests.

```
from __future__ import annotations
import sys
from xoa_core import (
    controller,
    types,
)
import asyncio
import json
from pathlib import Path
# XOA Converter is an independent module and it needs to be installed.
# ↪ via `pip install xoa-converter`
try:
    from xoa_converter.entry import converter
    from xoa_converter.types import TestSuiteType
except ImportError:
    print("XOA Converter is an independent module and it needs to be_
    ↪ installed via `pip install xoa-converter`")
    sys.exit()

PROJECT_PATH = Path(__file__).parent
OLD_2544_CONFIG = PROJECT_PATH / "old_2544_config.v2544"
OLD_2889_CONFIG = PROJECT_PATH / "old_2889_config.v2889"
PLUGINS_PATH = PROJECT_PATH / "test_suites"

async def subscribe(ctrl: "controller.MainController", channel_name:
    ↪ str, fltr: set["types.EMsgType"] | None = None) -> None:
    async for msg in ctrl.listen_changes(channel_name, _filter=fltr):
        print(msg)

async def main() -> None:
    # Define your tester login credentials
```

(continues on next page)

(continued from previous page)

```

my_tester_credential = types.Credentials(
    product=types.EProductType.VALKYRIE,
    host="10.20.30.40"
)

# Create a default instance of the controller class.
ctrl = await controller.MainController()

# Register the plugins folder.
ctrl.register_lib(str(PLUGINS_PATH))

# Add tester credentials into teh controller. If already added, it
→will be ignored.
# If you want to add a list of testers, you need to iterate
→through the list.
await ctrl.add_tester(my_tester_credential)

# Subscribe to test resource notifications.
asyncio.create_task(subscribe(ctrl, channel_name=types.PIPE_
→RESOURCES))

# Convert Valkyrie 2544 config into XOA 2544 config and run.
with open(OLD_2544_CONFIG, "r") as f:
    # get rfc2544 test suite information from the core's
→registration
    info = ctrl.get_test_suite_info("RFC-2544")
    if not info:
        print("Test suite is not recognized.")
        return None

    # convert the old config file into new config file
    new_data = converter(TestSuiteType.RFC2544, f.read())

    # you can use the config file below to start the test
    new_config = json.loads(new_data)

    # Test suite name: "RFC-2544" is received from call of c.get_
→available_test_suites()
    execution_id = ctrl.start_test_suite("RFC-2544", new_config)

    # The example here only shows a print of test result data.
    asyncio.create_task(
        subscribe(ctrl, channel_name=execution_id, fltr={types.
→EMsgType.STATISTICS})
    )

```

(continues on next page)

(continued from previous page)

```
# By the next line, we prevent the script from being immediately  
# terminated as the test execution and subscription are non-  
↪blockable, and they ran asynchronously,  
    await asyncio.Event().wait()  
  
if __name__ == "__main__":  
    asyncio.run(main())
```

2.2.6 Receive Test Result Data

XOA Core sends test result data (in JSON format) to your code as shown in the example below. It is up to you to decide how to process it, either parse it and display in your console, or store them into a file.

Listing 24: Receive test result data

```
async for stats_data in ctrl.listen_changes(execution_id, _filter=  
    ↪{types.EMsgType.STATISTICS}):  
    print(stats_data)
```

See also:

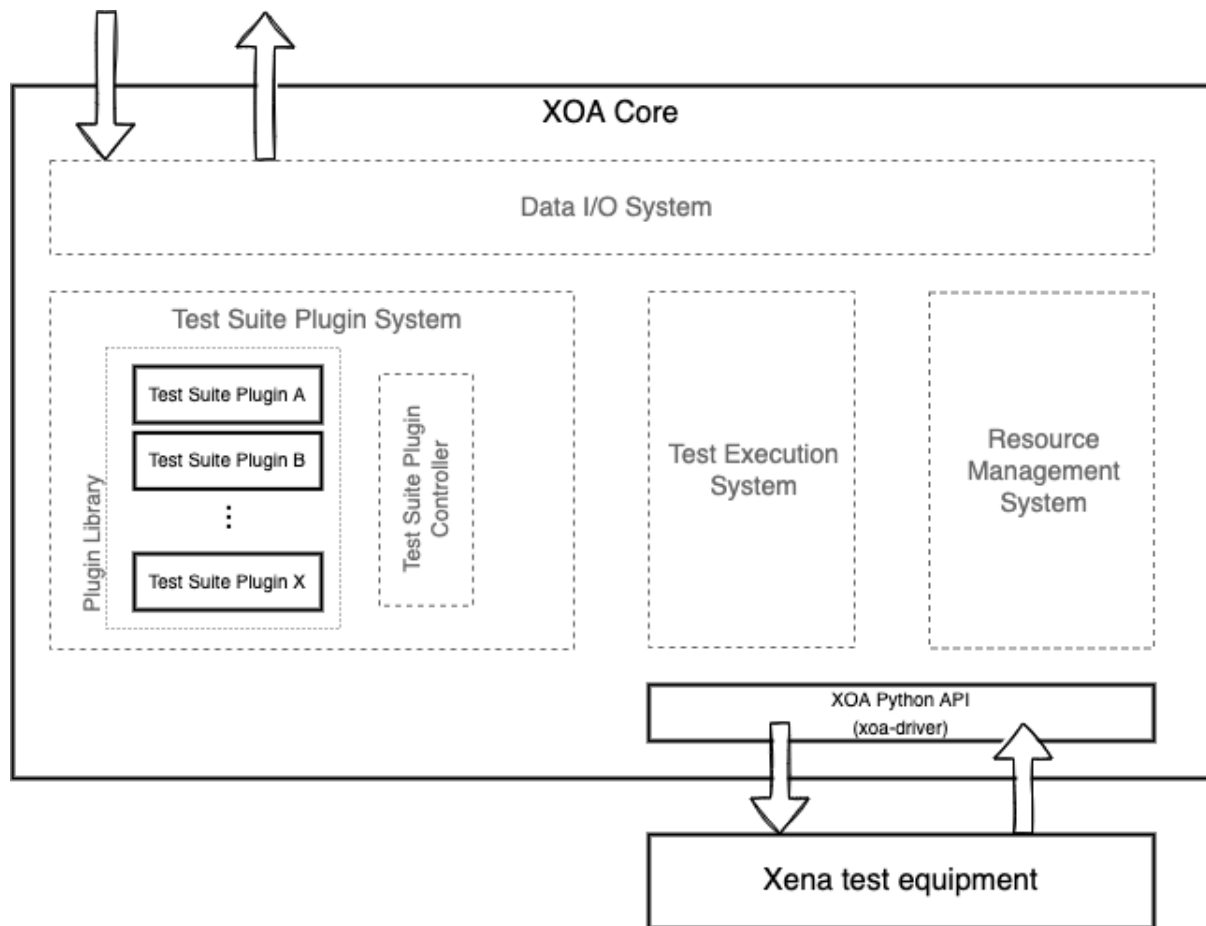
Read about *Test Result Types*

UNDERSTANDING XOA CORE

3.1 Introduction

The XOA Core is an asynchronous Python framework that can be represented by four subsystems:

1. *Resources Management System* lets you manage testers, i.e. add, remove, connect, disconnect, and view a list of available testers.
2. *Test Suite Plugin System* dynamically loads the test suites that are organized in a common structure, and exposes information of those test suites to you.
3. *Test Execution System* provides methods that let you control the test execution.
4. *Data IO System* let you subscribe to different messages generated by different test suites and subsystems so you can get test statistics/results, monitor test progress, receiver errors and warnings, etc.



3.2 Resources Management System

The key functionality is represented in managing and monitoring the state of known testers.

Available operations for users:

- Add testers
- Remove testers
- Connect to testers
- Disconnect from testers
- Get the list of available testers

You can find the corresponding APIs in [API Reference](#)

Under the hood, XOA Core uses the instance of `xoa_driver` library as a representation of the resource.

Note: `XOA Python API` (PyPI package name `xoa_driver`) is treated as a 3rd-party dependency, thus its source code is not included in XOA Core.

3.3 Test Suite Plugin System

All test suites are considered plug-ins by the XOA Core. You can freely choose which test suites to use. XOA Core dynamically loads test suites that are organized in a common structure, and exposes information of those test suites to you.

Available operations for users:

- Register a test suite into the plug-in library
- Get the name list of available test suites
- Get test suite information by its name

Users can register one or multiple test suite lookup folders in a test script by calling the method `register_lib(<lookup_path: str>)`.

3.3.1 Plugin Folder Structure

A test suite plug-in must have the structure below:

```
./my_test_suite
|
| - meta.yml
| - __init__.py
| - <any other modules defined by user>
```

`meta.yml` has a fixed structure as shown below, and is used as the entry point for the plug-in loading system. If the test suite folder doesn't contain this file, it will not be loaded by XOA Core.

Listing 1: `meta.yml` example

```
name: "RFC-2544[Frame Loss]" # Plugin name
version: "1.0" # Plugin current version
core_version: ">=1.0.0" # compatible to xoa-core version
author: # Optional list of authors
  - "ACO"
entry_object: "FrameLossTest" # class name of script entry point
data_model: "FrameLossModel" # class name of test suite data model
```

- `entry_object` must be inherited from an abstract class: `types.PluginAbstract`
- `data_model` must be a class of `Pydantic` model inherited from `pydantic.BaseModel`

Note: Be aware of imports during implementation of your plug-in. It is recommended to use relative import in your plug-in because the library paths in different user environments can be different, which makes it impossible for the plug-in code to run.

Important: Test suites are treated as an `asyncio.Task`. It means all heavy computational operations must be implemented with subprocess workers or threadings.

3.3.2 Plugin Example

We have developed a simple and executable test suite plug-in example doing RFC 2544 Frame Loss Test, and hope it help you get familiar with XOA Core.

You can find the source code of a [test suite plug-in example](#).

3.4 Test Execution System

XOA Core provides the following controlling methods of test suite execution:

- Start test
- Pause/continue test
- Stop test

3.4.1 Start Test

Use `execution_id = my_core_controller.start_test_suite(<plugin_name>, <suite_config_dict>)` to start a test and get the test ID as a returned value. With the test ID, you will be able to stop or pause the test.

<plugin_name> - must match the name from plugin's `meta.yml`.

<suite_config_dict> - must be a dictionary matching to the following structure:

Listing 2: Dictionary structure for <suite_config_dict>

```
{
  "username": "XOA",
  "port_identities": {
    "p0": {
      "tester_id": "2906f8d041e9fd07191d6a37ef5785b2",
      "tester_index": 0,
      "module_index": 1,
      "port_index": 4
    },
    ...
  },
  "config": TestSuiteModel<as dict>
}
```


If the test suite is successfully started, the function `start_test_suite` will return an `execution_id`, which can be used to control the test suite executions, or to subscribe to the outgoing messages from the test suite.

Note: A test suite will not start if its test resources are not registered in *Resource Manager*, or if one of its test resources is unavailable/disconnected.

3.4.2 Pause/Continue Test

Use `await my_core_controller.running_test_toggle_pause(<execution_id>)` to pause/continue a test.

User should use `await self.state_conditions.wait_if_paused()`, where the test suite should be paused/continued.

Note: To apply pause/continue action, a valid `execution_id` must be passed into the method.

3.4.3 Stop Test

Use `await my_core_controller.running_test_stop(<execution_id>)` to stop a test.

User should use `await self.state_conditions.stop_if_stopped()`, where the test suite should be stopped.

If the execution of `execution_id` exists, the test suite will be terminated.

3.5 Data IO System

XOA Core allows users to subscribe to different messages generated by different test suites and subsystems (ResourcesManager, ExecutorManager), so you can get test statistics/results, monitor test progress, receiver errors and warnings, etc.

3.5.1 Message Subscription

XOA Core provide two types of messages for you to subscribe to:

- Subsystem Type
- Test Execution Type

Subsystem Type Message

Subscribe to subsystem-type messages to receive messages exchanged between your code and the testers.

Listing 3: Subscribe to the messages from tester resource management subsystem.

```
async for msg in my_controller.listen_changes(<subsystem-type>):  
    # do whatever you want to the message
```

There are two types of subsystems that you can subscribe to:

Listing 4: Subsystem types

```
types.PIPE_EXECUTOR  
types.PIPE_RESOURCES
```

- `types.PIPE_EXECUTOR` messages tells information about the test executor.
- `types.PIPE_RESOURCES` messages tells information about the test resources, such as port reservation status, link sync status, tester connection status and so on.

Test Execution Type

A test execution can generate different types of messages, such as **test statistics**, **test progress**, **test state**, **errors**, and **warnings**. To subscribe to a specific type, you can use the `_filter` argument.

Listing 5: Subscribe to statistics of test execution

```
async for msg in my_controller.listen_changes(execution_id, _filter={  
    <filter_type>}):  
    # do whatever you want to the message
```

The `_filter` argument is an set of filter types. The first parameter of `_filter` argument is a mandatory identifier of the subsystem or the test suite execution. Available filters types are as shown below:

Listing 6: Available filters types

```
class EMsgType(Enum):  
    STATE = "STATE"  
    DATA = "DATA"  
    STATISTICS = "STATISTICS"  
    PROGRESS = "PROGRESS"  
    WARNING = "WARNING"  
    ERROR = "ERROR"
```

Note: `_filter` argument is optional. If it is not provided, all message types will be returned from this test suite execution.

The example below demonstrates how to subscribe to test resource notifications and test results.

```
from __future__ import annotations
from xoa_core import (
    controller,
    types,
)
import asyncio
import json
from pathlib import Path

PROJECT_PATH = Path(__file__).parent
XOA_CONFIG = PROJECT_PATH / "xoa_2544_config.json"
PLUGINS_PATH = PROJECT_PATH / "test_suites"

async def subscribe(ctrl: "controller.MainController", channel_name: str,
    fltr: set["types.EMsgType"] | None = None) -> None:
    async for msg in ctrl.listen_changes(channel_name, _filter=fltr):
        print(msg)

async def main() -> None:
    # Define your tester login credentials
    my_tester_credential = types.Credentials(
        product=types.EProductType.VALKYRIE,
        host="10.20.30.40"
    )

    # Create a default instance of the controller class.
    ctrl = await controller.MainController()

    # Register the plugins folder.
    ctrl.register_lib(str(PLUGINS_PATH))

    # Add tester credentials into teh controller. If already added, it
    # will be ignored.
    # If you want to add a list of testers, you need to iterate
    # through the list.
    await ctrl.add_tester(my_tester_credential)

    # Subscribe to test resource notifications.
    asyncio.create_task(subscribe(ctrl, channel_name=types.PIPE_
```

(continues on next page)

(continued from previous page)

```

→RESOURCES))

    # Load your XOA 2544 config and run.
    with open(XOA_CONFIG, "r") as f:

        # Get rfc2544 test suite information from the core's
→registration
        info = ctrl.get_test_suite_info("RFC-2544")
        if not info:
            print("Test suite RFC-2544 is not recognized.")
            return None

        # Test suite name: "RFC-2544" is received from call of c.get_
→available_test_suites()
        test_exec_id = ctrl.start_test_suite("RFC-2544", json.load(f))

        # The example here only shows a print of test result data.
        asyncio.create_task(
            subscribe(ctrl, channel_name=test_exec_id, fltr={types.
→EMsgType.STATISTICS})
        )

        # By the next line, we prevent the script from being immediately
        # terminated as the test execution and subscription are non-
→blockable, and they ran asynchronously,
        await asyncio.Event().wait()

if __name__ == "__main__":
    asyncio.run(main())

```

3.6 Test Result Types

XOA Core sends test result data (in json format) to your code as shown in the example below. It is up to you to decide how to process them, either [parse JSON into Python dictionary](#), display in your console, or store them into a file.

Listing 7: Receive test result data

```

async for stats_data in ctrl.listen_changes(execution_id, _filter=
→{types.EMsgType.STATISTICS}):
    print(stats_data)

```

There are three types of test result data (JSON format) that you receive from XOA Core.

- Live test result data

Every second, XOA queries statistics such as port TX and RX counters and sends them to you. The amount of this type of test result data can be large when your test duration is long.

- Intermediate test result data

If the test uses an iterative searching algorithm, such binary search in RFC 2544 Throughput Test and Back-to-Back Test, the result data after each searching step is called intermediate result because the searching is not yet complete. Intermediate results let you keep track of the searching steps.

- Final test result data

Final result data are the conclusion of a certain test iteration. For example, the throughput value for a certain frame size, the traffic latency value for a certain traffic rate with a certain frame size. This type of test result lets you analyze and verify the performance, conformance, and functionalities of your DUT/SUT.

To check the type of the test result data:

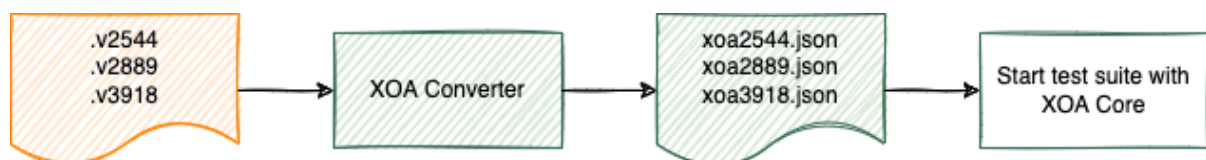
Listing 8: Check the type of the test result data

```
# The example here only shows a print of test result data.
async for stats_data in ctrl.listen_changes(execution_id, _filter=
    ↳ {types.EMsgType.STATISTICS}):
    if stats_data.is_final == True:
        # This is final result
    else:
        ...
```

3.7 Migrate from Valkyrie Test Suites

Xena's test suite applications have only been for Windows platform for a long time. Moving forward, all of Xena's existing and future test suites will be included in Xena OpenAutomation, which is not limited to Windows anymore.

We have developed a test configuration converter, [XOA Converter](#), to help users easily migrate their existing Windows test suite configurations (.v2544 for [Valkyrie2544](#), .v2889 for [Valkyrie2889](#), .v3918 for [Valkyrie3918](#), and .v1564 for [Valkyrie1564](#)) into XOA. The illustration below may help you understand the use flow.



For users of XOA who only uses the web GUI to create, import and run tests, there is no need to use this Python package, because XOA Core is already using this converter.

This converter is meant for those who want to integrate XOA test suites into their own Python environment without using the web GUI at all.

Note: The purpose of XOA Converter is **ONLY** to convert Xena Valkyrie test suit applications' configuration files into XOA's configuration files. Thus only four test suite types are supported by XOA Converter as the source config files.

API REFERENCE

4.1 Class MainController

GLOSSARY OF TERMS

API

Application Programming Interface.

IDE

An integrated development environment (IDE) is a software application that provides comprehensive facilities to computer programmers for software development.

Index Manager

An Index Manager manages the subport-level resource indices such as stream indices, filter indices, connection group indices, match term indices, length term indices, etc. It automatically ensures correct and conflict-free index assignment.

Module Manager

A Module Manager helps you access test modules. There is one Module Manager per tester.

Port Manager

A Port Managers helps you access test ports. There is one Port Manager per test module.

Resource Manager

XOA Python API HL-API provides an easy way to manage subtester test resources, including obtaining test resources and managing indices.

Test Resource

Test chassis, test module, and test port, both hardware and virtual are referred to as test resources. A user must have the ownership of a test resource before be able to perform testing.

TGA

Traffic Generation and Analysis.

Valkyrie1564

[Valkyrie1564](#) provides full support for both the configuration and performance test types described in Y.1564. It is installed together with ValkyrieManager and uses the same terminology. The simple intuitive GUI makes it easy to connect one or more ValkyrieCompact and/or ValkyrieBay chassis for testing Layer 2 and Layer 3.

Valkyrie2544

[Valkyrie2544](#) offers full support for the 4 test-types specified in RFC2544, and also lets you partially enable one or more test types. Valkyrie2544 supports different network

topologies and traffic flow directions on both Layer 2 and Layer 3, as well as both IPv4 and IPv6.

Valkyrie2889

[Valkyrie2889](#) is a free application for benchmarking the performance of Layer 2 LAN switches.

Valkyrie3918

[Valkyrie3918](#) provides an easy-to-use port configuration panel that lets you add and remove ports, and assign IP addresses and port roles. Ports from multiple ValkyrieBay and ValkyrieCompact chassis can be freely mixed.

XOA

Xena OpenAutomation

XOA CLI

XOA Command-Line Interface. Xena provides a rich set of CLI commands for users to administer test chassis for test automation. [Read more here.](#)

XOA Core

[Xena OpenAutomation Core](#) is an open test suite framework to execute XOA Test Suites as its plugins.

XOA Python API

The foundation of Xena OpenAutomation is its Python API (XOA Python API) that provides interfaces for engineers to manage Xena hardware and virtual test equipment.

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

A

API, [29](#)

I

IDE, [29](#)

Index Manager, [29](#)

M

Module Manager, [29](#)

P

Port Manager, [29](#)

R

Resource Manager, [29](#)

T

Test Resource, [29](#)

TGA, [29](#)

V

Valkyrie1564, [29](#)

Valkyrie2544, [29](#)

Valkyrie2889, [30](#)

Valkyrie3918, [30](#)

X

XOA, [30](#)

XOA CLI, [30](#)

XOA Core, [30](#)

XOA Python API, [30](#)