Application Note

R&S®AREG800A SCENARIO GENERATION USING PYTHON

Using the Open Simulation Interface and Python for creating dynamic radar scenarios

Products:

- R&S®AREG800A
- R&S®AREG8-24/81/81W
- R&S®QAT100

A. Meyer-Giesow | 1GP152 | Version 1 | 11.2025

https://www.rohde-schwarz.com/appnote/1GP152

Contents

1 Overview

The R&S®AREG800A is a powerful automotive radar echo generator, capable of over the air stimulation of automotive radar sensors with multiple dynamic radar objects. It can either be paired with R&S®AREG8 mmWave frontends for RF performance testing, or with the R&S®QAT100 antenna array for simulating ADAS scenarios.

The R&S®AREG800A has a scenario player built in for replaying predefined, dynamic scenarios. Scenarios come in the form of binary files containing a sequence of timestamped ASAM Open Simulation Interface (OSI) messages describing the radar objects that should be simulated. The R&S®AREG800A can replay these scenario files and provides convenient playback and remote-control functions, as well as automatic mapping of simulated objects to the connected RF frontends.

This application note presents background information and details about how to create said scenarios using Python and how to replay them on a R&S®AREG800A. Examples for scripting dynamic scenario in Python are provided as well.

The abbreviations “AREG” and “QAT” are used in this application note for the Rohde&Schwarz products R&S®AREG800A and R&S®QAT100.

2 Introduction

Objects simulated by the AREG represent point targets, with each object having an individual range, Doppler speed, radar cross-section (RCS), and azimuth value. The AREG can simulate up to eight objects on a single mmWave frontend and up to four objects on a single QAT. Multiple frontends can be controlled by a single AREG. This can be used to create setups with an increased field of view, or to stimulate multiple sensors at once.

To test radar sensors and associated systems in dynamic scenarios, the AREG can simulate moving objects. Unlike in static object mode, no manual object assignment to individual frontends or QAT segments is necessary. Instead, objects are provided to the AREG as a global object list. They are automatically mapped to a suitable frontend and channel. Objects that cannot be simulated will be listed as invalid.

Each dynamic object is defined by individual range, azimuth, Doppler, and RCS values. The AREG simulates dynamic scenarios by processing timestamped object lists, with the new object updates becoming active at a user defined timestamp. These object lists are always defined in the ASAM Open Simulation Interface (OSI) message format.

There are two ways of getting OSI messages to the AREG:

- The scenario mode allows to do open loop testing by replaying **.osi* scenario files. These scenario files contain a timestamped list of OSI messages that are played back by the AREG according to their timestamps. This application note deals with creating these scenario files by using Python scripts.
- The real time control interface can be used to do open or closed loop testing. Instead of using **.osi* files like the scenario mode, the object list is updated by streaming the same OSI messages to the AREG interface. The real time control interface can also be used by a third-party environment simulation. The real time interface is not covered in this application note.

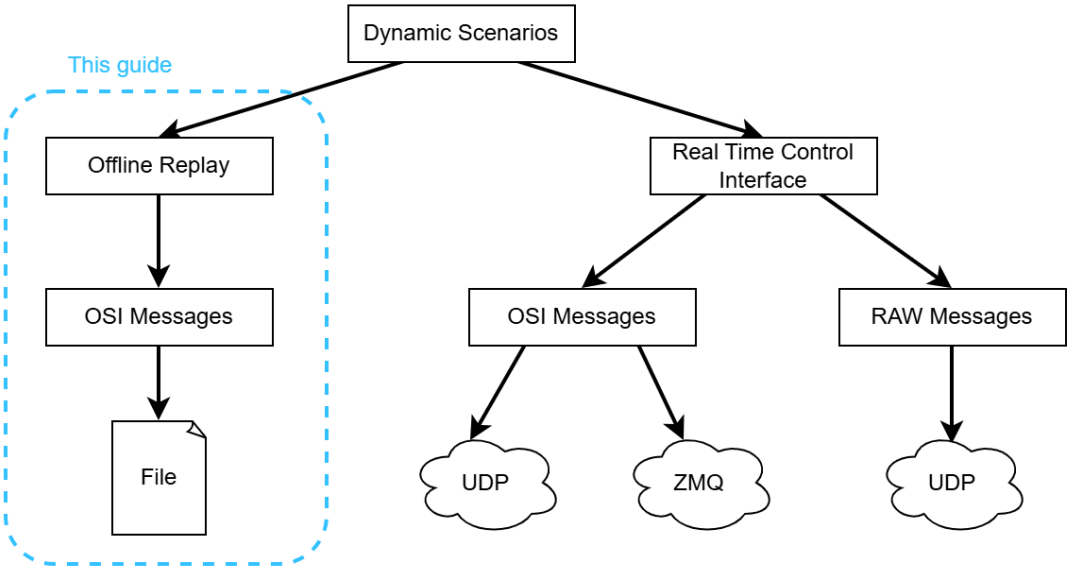

Figure 1: Overview of different modes and formats for generating dynamic scenarios on the AREG800A.

3 Open Simulation Interface

The Open Simulation Interface defines interfaces between components of distributed simulations, focusing on environmental perception for automated driving functions. It provides an object-based environment description and uses Google Protocol Buffers for serialization. Key top-level messages include:

- *GroundTruth*: complete simulated environment in the global coordinate frame over time.
- *SensorView*: input to sensor models, derived from *GroundTruth*; environment expressed in the virtual sensor frame.
- *SensorData*: emulates real sensor output, may be derived from *GroundTruth*, *SensorView*, or *FeatureData*. Used by automated driving functions, sensor models, and fusion.
- *FeatureData*: detected features in a sensor's reference frame, derived from *GroundTruth*. Used for object detection and feature fusion.

The AREG uses the *SensorData* interface. *SensorData* includes a timestamp for when the update should occur and a sensor ID identifying the source sensor. The objects to be simulated are carried in the *FeatureData* interface.

3.1 AREG800A OSI Message Format

The AREG expects protobuf-serialized OSI *SensorData* messages. Each *SensorData* message must include a timestamp (the time the update should occur), and *sensor_id* (identifying which of the sensors configured on the AREG should be used). The payload is a *FeatureData* message containing *RadarDetectionData*, which holds one or more *RadarDetection* entries. Figure 2 shows the fields populated in the AREG OSI message format.

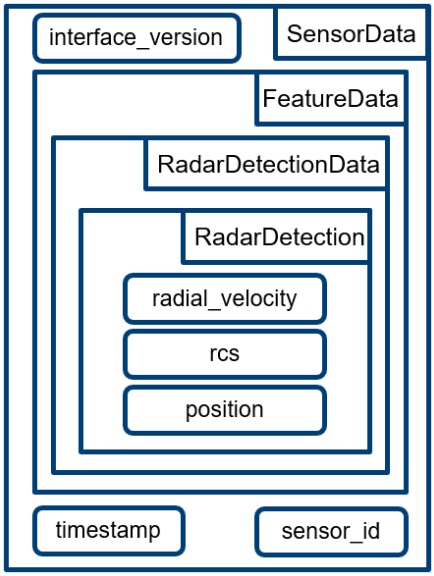

Figure 2: OSI message format for the AREG800A.

A *RadarDetection* provides position (in the sensor's coordinate frame), *radial_velocity*, and *rcs* (radar cross section). Each *RadarDetection* entry corresponds to a single object simulated by the AREG.

Table 1 describes the message fields in more detail, including data types and units. Note that the timestamp of the OSI message is split into a second and nanoseconds part.

Parameter		Data type	Description
interface_version (optional)	**major**	uint32	Major version number.
	minor	uint32	Minor version number.
	patch	uint32	Patch version number.
timestamp	**seconds**	int64	The number of seconds since the start of the simulation Unit: s
	nanos	uint32	The number of nanoseconds since the start of the last second. Unit: ns
sensor_id		uint64	The ID of the sensor.
position	**distance**	double	The radial distance. Unit: m
	azimuth	double	The azimuth (horizontal) angle. Counter-clockwise is positive. Unit: rad
	elevation	double	The elevation (vertical) angle. Up is positive. Unit: rad
radial_velocity		double	Radial velocity of the detection, positive when moving away from the sensor.. Unit: m/s
rcs		double	The radar cross section (RCS) of the radar detection. Unit: dBm^2 Conversion from m^2 to dBm^2: $RCS_{dBm2} = 10 * \log_{10}(RCS_{m2})$

Table 1: Field description for the AREG800A OSI message format.

3.2 AREG800A Scenario File Binary Format

OSI messages are serialized using Google Protocol Buffers (protobuf). On a binary level, they are representations of top-level OSI messages such as the *SensorData* message used by the AREG. The message structure is defined by the OSI *.proto* files, while the binary encoding follows the protobuf wire-format specification. The protobuf compiler (protoc) produces language specific bindings from the OSI *.proto* files. These bindings implement serialization and deserialization, removing the need to handle the binary layout manually.

To generate an OSI message for the AREG, generate language-specific bindings with the protobuf compiler, populate the message fields of the *SensorData* message as described in the AREG800A OSI message format, and invoke the library's serialization method. When writing to a file, each message is prefixed with a with a length field containing the size of the following message in bytes.

An OSI scenario file is therefore a list of individual OSI messages containing one or multiple radar objects, with increasing timestamps. The AREG simulates the list of radar objects described in each OSI message. A new object list becomes active at each new timestamp.

3.3 Simulating Objects with Constant Echo Power

OSI messages specify objects via RCS only, so echo power varies with range: nearer objects yield higher echo power. Some tests need a constant echo power regardless of range. AREG supports this in static mode by specifying a signal attenuation value directly. When using OSI messages, the same effect can be achieved by adjusting RCS over time to offset path loss: increase RCS as range increases and decrease it as range decreases.

AREG relates attenuation, RCS, range, and center frequency by the following formula:

$$\mathrm{RCS} = \frac{\lambda^2}{4\pi}\frac{R^4}{A^4}\frac{1}{Att_0}$$

with $\lambda = c/f$, $c = 299700000\ \mathrm{m/s}$, f the radar center frequency, R the object distance, A the air gap between sensor and AREG antennas, and Att_0 the object attenuation (linear, not dB).

By setting Att_0 constant and computing RCS for each timestamp from the current range, the echo power then remains constant across the scenario. An example script for generating a constant echo power range sweep is supplied together with this application note.

4 Using Python for Scenario Generation

The following Section describes how to generate an OSI file for a simple range sweep scenario, using Python and the publicly available Python bindings for the OSI interface.

4.1 Prerequisites

To generate OSI messages using Python, the official OSI Python package containing the protobuf bindings can be used. The minimum required OSI version is 3.5.0. The installation process differs depending on which operating system is used. Official instructions on how to install the Python and C++ bindings are available in the ASAM OSI documentation:

- Windows: https://opensimulationinterface.github.io/osi-antora-generator/asamosi/latest/interface/setup/installing_windows_python.html
- Linux: https://opensimulationinterface.github.io/osi-antora-generator/asamosi/latest/interface/setup/installing_linux_python.html

4.2 Example Script Walk Through

To generate OSI messages, we need to import the *SensorData* class from the Python OSI package. The *SensorData* interface is the base for communication with the AREG800A. We also need the *struct* package to pack the serialized OSI messages into a binary file.

```
from osi3.osi_sensordata_pb2 import SensorData
import struct
```

Next, we define helper functions to deal with the OSI messages. the *OsiRadarSensor* class creates an OSI message as expected by the AREG. For each update, an *OsiRadarSensor* is initialized with a sensor ID and timestamp. The timestamp defines the scenario time at which this OSI message becomes active. Radar objects are added with the *add_detection* function. Internally, the *add_detection* function adds a sensor detection to the *SensorData* message and populates the field necessary for the AREG. To simulate multiple objects, call the *add_detection* function multiple times. Finally, the message can be serialized with the *serialize_to_byte_msg* function. The serialization of the OSI message is provided by the Python OSI package and uses Google Protobuf.

The *generate_osi_msg* function does all the above for a single radar object:

```
class OsiRadarSensor:
    """Wrapper for OSI Python interface"""
    def __init__(self, sensor_id=1, timestamp_sec=0, timestamp_nanos=0):
        self.sensordata = SensorData()
        self.sensordata.sensor_id.value = sensor_id
        self.sensordata.timestamp.seconds = timestamp_sec
        self.sensordata.timestamp.nanos = timestamp_nanos
        self.radar_sensor = self.sensordata.feature_data.radar_sensor.add()

    def add_detection(self, distance, azimuth, radial_velocity, rcs):
        detection = self.radar_sensor.detection.add()
        detection.position.distance = distance
        detection.position.azimuth = azimuth
        detection.radial_velocity = radial_velocity
        detection.rcs = rcs

    def serialize_to_byte_msg(self):
        bytes_buffer = self.sensordata.SerializeToString()
        return bytes_buffer


def generate_osi_msg(ts_sec, ts_nanos, range, azimuth, velocity, rcs):
    """
    Generates OSI message for the specified target.

    Returns:
        buffer containing serialized OSI message
    """
    rad = OsiRadarSensor(1, ts_sec, ts_nanos)
    rad.add_detection(range, azimuth, velocity, rcs)
    msg = rad.serialize_to_byte_msg()
    return msg
```

We also need a function that generates positions for the object's movement. In this example, the *range_sweep* function calculates the positions of an object moving towards the radar sensor at a specified speed. After the *range_sweep* generator function is initialized with the desired parameters, we can then iterate on it to obtain serialized OSI messages that we can write to the *.osi* scenario file.

```
def range_sweep(interval, range_start, range_stop, vel, rcs, az):
    """
    Yields OSI messages for an object moving towards radar at constant velocity

    Parameters:
        interval: interval at which the movement should be updated in seconds
        range_start: start of range sweep in m
        range_stop: stop of range sweep in m
        vel: radial velocity of target in m/s
        rcs: radar-cross section of target in dBsm
        az: azimuth of target in radians (positive turns counter-clockwise)

    Returns:
        serialized OSI message
    """
    if range_start < range_stop:
        raise ValueError("range_start must be greater than range_stop")
    if vel > 0.0:
        raise ValueError("Radial velocity must be negative for a target moving
towards the radar")
    if interval < 0.01:
        raise ValueError("Update interval for scenarios should be >= 10ms")

    timestamp_sec = 0
    timestamp_nanos = 0
    range_step = vel * interval
    curr_range = range_start
    while curr_range >= range_stop:
        msg = generate_osi_msg(timestamp_sec, timestamp_nanos, curr_range, az,
vel, rcs)
        curr_range += range_step
        timestamp_nanos += int(interval * 1e9)
        if timestamp_nanos >= 1000000000:
                timestamp_sec += 1
                timestamp_nanos = 0
        yield msg
```

In the final step, everything is put together. We set the scenario parameters, allocate a buffer to write the OSI messages to, and initialize the *range_sweep* position generator. We then iterate over the position generator until it reaches its stop condition and append each OSI message to the buffer. Finally, the filled buffer is written to a file.

```
if __name__ == "__main__":
    # scenario settings
    update_interval = 0.1 # scenario update rate in seconds
    start = 120       # m
    stop = 20.0       # m
    velocity = -10.0 # m/s
    rcs = 10          # dBsm
    azimuth = 0.0     # rad

    buf = bytes()
    osi_msg = range_sweep(update_interval, start, stop, velocity, rcs, azimuth)
    while True:
        try:
            next_msg = next(osi_msg)
        except StopIteration:
            print('End of scenario')
            break
        buf += struct.pack("<L", len(next_msg)) + next_msg

    f = open("range_sweep_example.osi", "wb")
    f.write(buf)
    f.close()
```

The complete script can be found in the additional files provided with this application note.

5 Running a Scenario on the AREG800A

Once an OSI scenario file has been generated, it must be transferred to the AREG. This can be done via SD Card or USB flash drive. A more convenient method is to transfer the OSI file to the AREG via network (SMB or FTP). For more details see the chapter “Transferring files from and to the instrument” in the instrument manual (link to online manual).

If the scenario generation/instrument control is done using Python, you can also use the *RsInstrument* Python package, which contains a convenient function for file transfer directly out of your Python script. See the Related Links section for more details.

To enable the scenario replay mode on the AREG:

1. Go to the Operation Setup tile and select “Mode > Dynamic” and “Data Source > Scenario”.

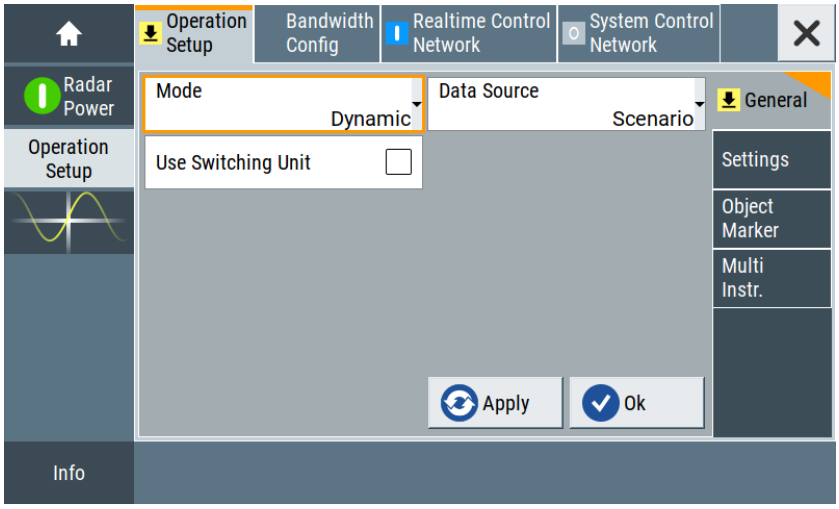

Figure 3: Operation Setup of the AREG800A set to scenario replay mode.

2. On the home screen, go to “Radar Objects > Scenario”.

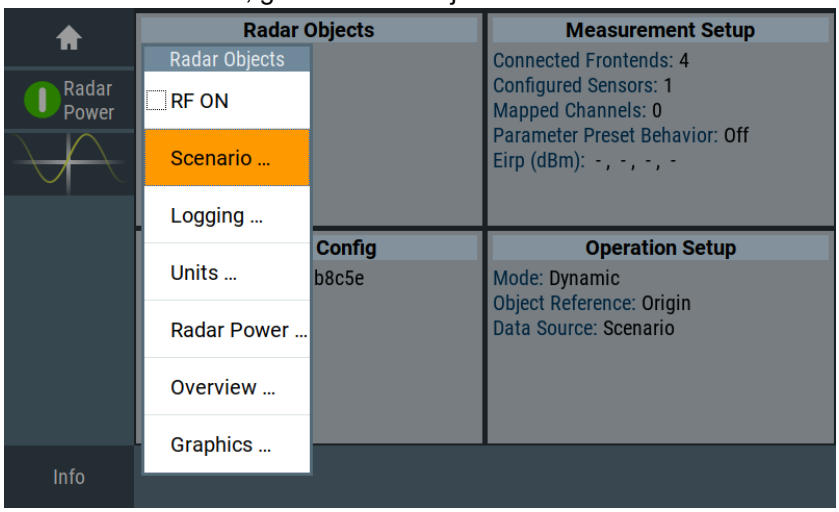

Figure 4: Access to the scenario player via the home screen.

3. The scenario player screen will open and provide options to select and play an OSI file on the AREG.

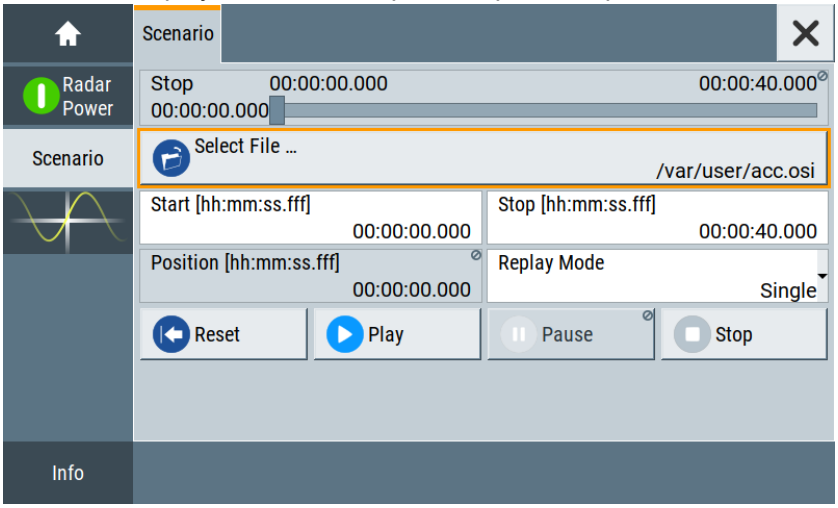

Figure 5: Scenario player screen of the AREG800A.

For more details on the scenario player, see the chapter “Scenario settings” in the user manual (link to online user manual). All actions, including starting the replay can be automated via SCPI commands.

5.1 Monitoring Simulated Objects in the Overview Screen

The AREG overview screen provides real-time monitoring of scenario inputs and frontend mappings. It displays the received OSI objects within a polar view. The AREG evaluates each object against the current frontend setup, considering simulation restraints like field of view and minimum distance, and marks non-simulatable objects as invalid.

In Figure 6, three objects are received: two are simulated (blue) and one is invalid (gray) because it is outside the QAT100 frontend’s field of view; the shaded orange wedges indicate the available frontend sectors and the red cross marks the sensor position.

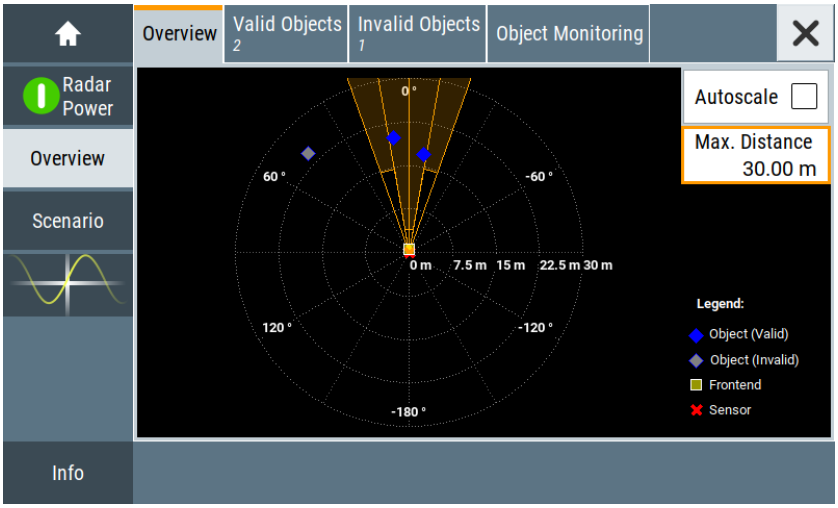

Figure 6: Object overview screen during scenario replay.

6 Related Links

ASAM OSI website: https://www.asam.net/standards/detail/osi/

Python OSI repository: https://github.com/OpenSimulationInterface/open-simulation-interface

OSI user guide: https://opensimulationinterface.github.io/osi-antora-generator/asamosi/latest/specification/index.html

AREG800A online manual: https://www.rohde-schwarz.com/webhelp/AREG_HTML_UserManual_en/Content/welcome.htm

RsInstrument Python Package: https://pypi.org/project/RsInstrument/

File transfer with RsInstrument: https://rsinstrument.readthedocs.io/en/latest/StepByStepGuide.html#pc-instrument

7 Ordering information

Designation	Type	Order No.
Base unit		
Automotive radar echo generator	R&S®AREG800A	1437.4400.02
R&S®QAT100 advanced antenna array		
Advanced antenna array, from 76 GHz to 81 GHz	R&S®QAT100	1341.0004.02
mmWave remote frontends		
76 GHz to 81 GHz, single antenna, 5 GHz RF bandwidth	R&S®AREG8-81WS	1437.9153K02
76 GHz to 81 GHz, two antennas, 5 GHz RF bandwidth	R&S®AREG8-81WD	1437.9160K02

Rohde & Schwarz

The Rohde & Schwarz electronics group offers innovative solutions in the following business fields: test and measurement, broadcast and media, secure communications, cybersecurity, monitoring and network testing. Founded more than 80 years ago, the independent company which is headquartered in Munich, Germany, has an extensive sales and service network with locations in more than 70 countries.

www.rohde-schwarz.com

Certified Quality Management
ISO 9001

Rohde & Schwarz training

www.rohde-schwarz.com/training

Rohde & Schwarz customer support

www.rohde-schwarz.com/support

PAD-T-M: 3572.7186.00/02.00/EN

1GP152 | Version 1 | 11.2025
Application Note | R&S®AREG800A Scenario Generation Using Python
Data without tolerance limits is not binding | Subject to change

www.rohde-schwarz.com