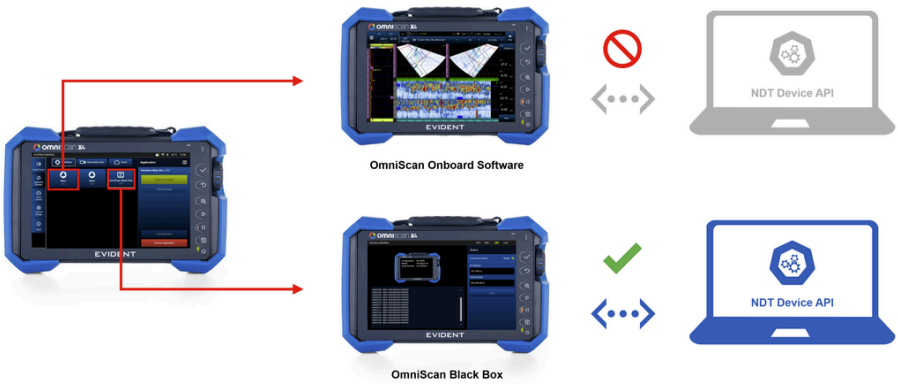
R&D Note | FMC Capture Using the OmniScan™ Black Box App

OmniScan Black Box

What is the OmniScan Black Box?

The OmniScan Black Box is an application used with the OmniScan™ X4 phased array flaw detector. This app allows for **computer control** of the OmniScan X4 unit using data acquisition software such as WeldSight™, AeroView™, or other custom software. It enhances the device's capabilities by enabling advanced data collection and analysis and facilitating the integration of the OmniScan X4 into broader inspection ecosystems.



What is the NDT Device API?

The NDT Device API (application programming interface) is an open programming interface designed to facilitate the integration and computer control of Evident NDT instruments within an inspection ecosystem. This API allows for easy device abstraction and connectivity, enabling different instruments to communicate and be controlled remotely.

What is the difference between OmniScan Black Box app and the NDT Device API?

The NDT Device API provides the underlying protocol and commands for communication and integration, while the OmniScan Black Box is an application that leverages this API to enable a computer to control an OmniScan X4 unit's data acquisition. In other words, the OmniScan Black Box App enables any external software to communicate with the OmniScan X4 using the language defined by the NDT Device API, acting as a translation layer.

Full Matrix Capture (FMC)

What about FMC acquisition?

One of the advanced features offered by the NDT Device API through the OmniScan Black Box app is the ability to perform FMC elementary A-scan capture with an OmniScan X4 unit. This document describes an example use case and the associated steps required to realize an FMC capture using an OmniScan X4 **64:128PR** unit with OmniScan Black Box app and a Python environment.

> Note that OmniScan X4 units, through the OmniScan Black Box app, currently support only FMC as an advanced firing pattern. Conventional UT and PAUT acquisitions are also supported. Please reach out to our NDT Device API advanced support for additional documentation and guidelines.

> The code presented below is provided as is, with no warranty, and should not be considered as a formal software implementation.

Set Up Your OmniScan Unit and PC

- Connect your OmniScan unit and your PC together using a Gigabit Ethernet cable.
- Start your OmniScan X4 unit and launch the OmniScan Black Box app.
- On the OmniScan Black Box app: Set the IP Address to 192.168.0.2 and the Subnet Mask to 255.255.255.0
- On your PC: Set the IP Address to 192.168.0.1 and the Subnet Mask to 255.255.255.0

> The OmniScan Black Box app will have the **Server** role and your PC the **Client** role.

> The OmniScan X4 unit and the PC must be on the same subnet. Accessing the OmniScan through a routed network or from outside its subnet is not supported yet.

Set Up Your Development Environment

First, we highly recommend using a Python IDE (integrated development environment) such as VSCode, PyCharm, or Spyder to develop and debug a Python application effectively.

Communication with the NDT Device API through the OmniScan Black Box app is achieved using the **ZeroMQ** library. Hence, you will need the **pyzmq** Python bindings for ZeroMQ. These can be easily installed using a Python package manager such as **pip**:

```
pip install pyzmq
```

The server that receives and parses commands (OmniScan Black Box) uses a ZeroMQ socket with the **Request-Reply (REQ-REP)** pattern on port **5556**. This means that to communicate with the server, you (the client) need to use a ZeroMQ **REQ** socket connected to port **5556** of the server. The **JSON-RPC** standard is used for communication, with version 2.0 currently in use.

Once the acquisition is started, the server that sends the data stream (OmniScan Black Box) uses a ZeroMQ socket with the **Publish-Subscribe (PUB-SUB)** pattern on port **6712**. This means that to receive data from the server, you (the client) need to use a ZeroMQ **SUB** socket connected to port **6712** of the server.

For the FMC acquisition example provided below, we will also use the well-known **matplotlib** and **numpy** libraries:

```
pip install matplotlib numpy
```

Test the Connection

To test the connection with the instrument (server), we will first send a simple command, ***get_state***, and observe if a response is sent back to the PC (client).

The JSON-RPC formatted command is as follows:

```
{
    "jsonrpc": "2.0",
    "id": 1,
    "method": "get_state"
}
```

We will now send the command and listen for a response using a ZeroMQ REQ socket through a simple Python script:

```
import zmq

command = '{"jsonrpc": "2.0","id": 1,"method": "get_state"}'

context = zmq.Context()
socket = context.socket(zmq.REQ)
socket.connect(f"tcp://192.168.0.2:5556")

socket.send_string(command)
response = socket.recv_string()

print(response)
```

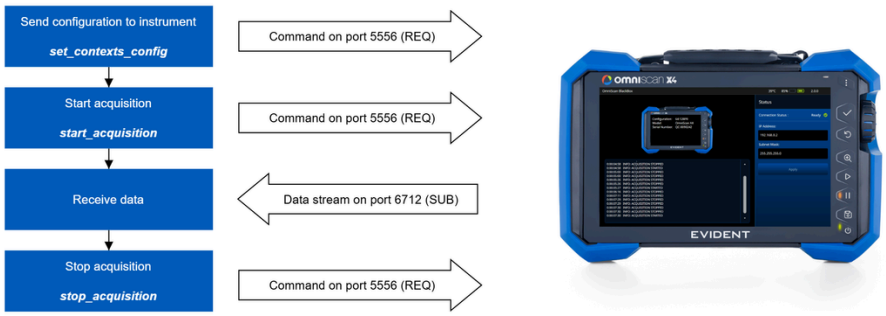
You should receive the following response:

```
{"jsonrpc": "2.0", "result": {"acquisition_state":"stopped","provisioning_state":"assigned"},"id": 1 }
```

If you do not receive a response, there is probably something wrong with your configuration.

Perform an FMC Acquisition

Now that you understand the basics of interacting with the instrument, we will guide you through performing an FMC acquisition. You will need to send the pulsing and receiving sequence configuration to the instrument using the ***set_contexts_config*** command. Then, start the acquisition using the ***start_acquisition*** command, receive data by listening on port 6712 with a SUB socket, and stop the acquisition using the ***stop_acquisition*** command (see figure below).



set_contexts_config command configuration

The configuration of the ***set_contexts_config*** command can be time-consuming, as it requires fully defining the FMC acquisition sequence. This command manages the definition of all necessary focal laws along with additional parameters. To set it up, we first define the negative square pulse within the **waveforms** array. Then, we specify the acquisition sequence within the ***pulse_receive_config*** array, where successive elements emit while all elements receive for FMC (see the example structure below). The command can be easily generated through a short script. For conciseness, an example configuration for a 64-element, 5 MHz linear array is provided: set_contexts_config_FMC_X4.json

> Note that OmniScan X4 units, through the OmniScan Black Box app, currently support only FMC as an advanced firing pattern. Conventional UT and PAUT acquisitions are also supported. Please reach out to our NDT Device API advanced support for additional documentation and guidelines.

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "set_contexts_config",
  "params": {
    "pulse_receive_pa":{
        "waveforms":[
          {
              "id":0,
              "waveform_steps":[  // Define a square negative pulse
                {
                    "id":0,
                    "amplitude":5,  // Amplitude is set to 80V
                    "width":100     // Pulse width in ns, usually set to a half-cycle (0.5/frequence)
                }
              ]
          }
        ],
        "pulse_receive_config":[
          {  // First sequence of the FMC in which probe element 1 is pulsing
              "elementary_ascan_data_enabled":true,
              "id":0,
              "length":1600, // A-Scan length of the first sequence in samples
              "pulsers":[
                {
                    "delay":0, // No delay in this case
                    "element_index":0, // Index start at 0, corresponding to element 1
                    "id":0,
                    "pulser_waveform_id":0 // Square pulse used
                }
              ],
              "receivers":[    // Defining reception on all elements
                {
```

```
                    "element_index":0,  // Reception on probe element 1 first
                    "gain":0,    // Gain in recepetion set to 0 in this case
                    "id":0
                },
                {...}  // Repeat for each receiving elements
            ],
            "start":0,
            "elementary_ascan_decimation":1
        },
        { // Second sequence of the FMC in which probe element 2 is pulsing
            "elementary_ascan_data_enabled":true,
            "id":1,
            "length":1600,
            "pulsers":[
                {
                    "delay":0,
                    "element_index":1, // Pulsing with second element of the probe
                    "id":0,
                    "pulser_waveform_id":0
                }
            ],
            "receivers":[
                {
                    "element_index":0,  // Reception on probe element 1 first
                    "gain":0,
                    "id":0
                },
                {...} // Repeat for each receiving elements
            ],
            "start":0,
            "elementary_ascan_decimation":1
        },
        {...}  // Repeat for each emitting elements
      ]
    }
  }
}
```

Assuming you stored your ***set_contexts_config*** command configuration in a separate JSON file (such as `set_contexts_config_FMC_X4.json`), it can then be sent using a ZeroMQ REQ socket:

```python
import zmq
import json

with open('set_contexts_config_FMC_X4.json', 'r') as file:
    command = json.load(file)

command = json.dumps(command)

context = zmq.Context()
socket = context.socket(zmq.REQ)
socket.connect(f"tcp://192.168.0.2:5556")

socket.send_string(command)
response = socket.recv_string()

print(response)
```

start_acquisition command configuration

The ***start_acquisition*** command structure is very simple:

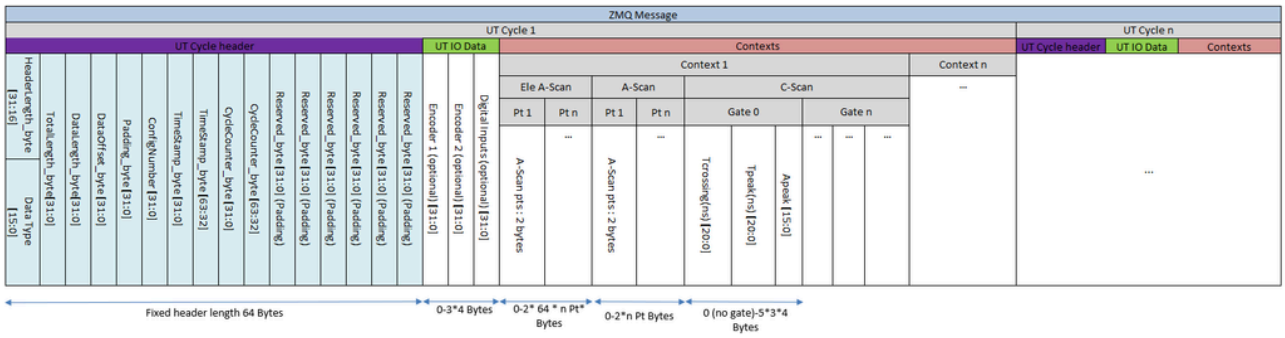
```
{
    "jsonrpc": "2.0",
    "id": 1,
    "method": "start_acquisition"
}
```

and it can be sent using a ZeroMQ REQ socket:

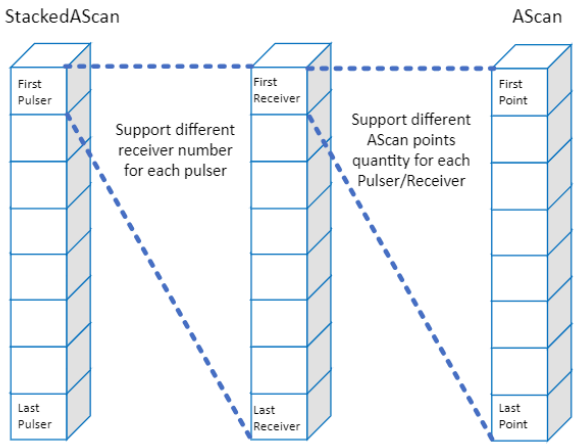
```
import zmq

command = '{"jsonrpc": "2.0","id": 1,"method": "start_acquisition"}'

context = zmq.Context()
socket = context.socket(zmq.REQ)
socket.connect(f"tcp://192.168.0.2:5556")

socket.send_string(command)
response = socket.recv_string()

print(response)
```

Receive the Data Stream

The data stream (ZMQ Message) sent on port 6712 and received through a ZeroMQ SUB socket always has the same structure (see illustration below). Each acquisition cycle (a cycle is the execution of all the configured contexts, or focal laws) is formatted with a fixed header of 64 bytes, followed by the collected data.



In the case of an FMC capture, all A-scans are stacked in a single vector (see illustration below) for each acquisition cycle.



Knowing this data stream structure, you can create a ZeroMQ SUB socket to receive messages, connect it to the server, and subscribe to all incoming messages. Then, you can receive the data stream, which can be converted into a numpy array for easier manipulation. An example code is provided below, in which we plot the first A-scan of the FMC (corresponding to element 1 emitting and receiving). Note that we also added a timeout to handle any transmission errors.

```python
import numpy as np
import zmq
import matplotlib.pyplot as plt

# Initialize ZeroMQ context and socket
context = zmq.Context()
socket_data = context.socket(zmq.SUB)
socket_data.connect("tcp://192.168.0.2:6712")
socket_data.subscribe("")

# Set the socket option for receive timeout
socket_data.RCVTIMEO = 5000  # Timeout in milliseconds

try:
    # Attempt to receive data
    databuffer = socket_data.recv()

    # Extract data from buffer
    header_size = 64
    fmc = np.frombuffer(databuffer[header_size:], dtype=np.int16)

    # Plot the data
    plt.plot(fmc[:3999])
    plt.ylabel('First Elementary Ascan')
    plt.show()
except zmq.Again:
    print("No data received within the timeout period.")
```

stop_acquisition **command configuration**

After retrieving your data, the ***stop_acquisition*** command can be invoked, again, very simply:

```
{
    "jsonrpc": "2.0",
    "id": 1,
    "method": "stop_acquisition"
}
```

and can be sent using a ZeroMQ REQ socket:

```python
import zmq

command = '{"jsonrpc": "2.0","id": 1,"method": "stop_acquisition"}'

context = zmq.Context()
socket = context.socket(zmq.REQ)
socket.connect(f"tcp://192.168.0.2:5556")

socket.send_string(command)
response = socket.recv_string()

print(response)
```

Additional Resources

- FMC Principles: FMC-TFM Basic Principles
- ZeroMQ Documentation and Ressources: ZeroMQ

Note edited on March 21, 2025.