



esoc

European Space Operations Centre
Robert-Bosch-Strasse 5
D-64293 Darmstadt
Germany
T +49 (0)6151 900
F +49 (0)6151 90495
www.esa.int

DOCUMENT

Development Guide for NMF Apps

Prepared by César Coelho
Reference
Issue
Revision
Date of Issue
Status
Document Type TN
Distribution



Table of contents:

| | | |
|----------|---|-----------|
| 1 | INTRODUCTION | 3 |
| 2 | REFERENCES | 3 |
| 2.1 | Referenced Documents | 3 |
| 3 | NMF APPS | 4 |
| 4 | BUILD THE NANOSAT MO FRAMEWORK | 4 |
| 5 | DEVELOP AN NMF APP | 5 |
| 5.1 | Creating a project | 5 |
| 5.2 | Initializing the NanoSat MO Connector | 5 |
| 5.3 | Monitor and Control integration | 6 |
| 5.3.1 | M&C Adapters | 7 |
| 5.3.2 | Pushing a Parameter Value | 10 |
| 5.3.3 | Raising an Alert | 10 |
| 5.3.4 | Reporting Action Execution Progress | 11 |
| 5.4 | Accessing the Platform services | 13 |
| 5.4.1 | Camera | 13 |
| 5.4.2 | GPS | 14 |
| 5.4.3 | AutonomousADCS | 15 |
| 5.4.4 | Software-defined Radio | 17 |
| 5.4.5 | Optical Data Receiver | 18 |
| 5.4.6 | Magnetometer | 18 |
| 5.4.7 | Power Control | 18 |
| 6 | TESTING THE APP | 20 |
| 6.1 | Run the NMF App in the SDK environment | 20 |
| 6.2 | Automatic deployment of the NMF App | 21 |
| 6.3 | Project with extra external dependencies | 21 |
| 7 | HANDS-ON ACTIVITIES | 22 |
| 7.1 | Activity 1 | 22 |
| 7.2 | Activity 2 | 22 |
| 8 | FAQ | 23 |
| 8.1 | How to convert from a Java primitive data type to a MAL data type and vice versa? | 23 |
| 8.2 | How to drop the COM Archive database table at start up? | 23 |
| 8.3 | How to change the transport layer? | 23 |
| 9 | MAL ATTRIBUTE DATA TYPES | 24 |



1 INTRODUCTION

This document explains how to develop an app using the NanoSat MO Framework. The developer is welcomed to read the “Quick Start” guide before reading this document. This document was produced as part of the NanoSat MO Framework Software Development Kit (SDK).

It is very important to always gather feedback from the developers in order to improve the framework. For that reason, an area online was created where developers are encouraged to report bugs, problems, suggest new ideas or improvements:

<https://github.com/esa/nanosat-mo-framework/issues>

The NanoSat MO Framework is available online on GitHub under an open source licence. Additionally, it will also be available on Maven Central in order to facilitate the project dependencies resolution.

The NanoSat MO Framework implementation was developed in Java and as a minimum requirement, Java version 8 is necessary in order to run the software. The developer is suggested to use Netbeans IDE during the software development process.

2 REFERENCES

2.1 Referenced Documents

| Ref. | Title | Code | Issue | Date |
|-------|---|-------------------|-------|------------|
| [RD1] | NanoSat MO Framework: Achieving On-board Software Portability | <<To Be Defined>> | | May 2016 |
| [RD2] | CCSDS Mission Operations Services on OPS-SAT | IAA-B10-1301 | | April 2015 |
| [RD2] | NanoSat MO Framework – Quick Start | | | Feb 2015 |

3 NMF APPS

An NMF App is an on-board software application designed to take advantage of the NMF. Upon start-up, it will register itself in the Central Directory service for visibility from ground and unregister before terminating gracefully. Additionally, it can connect and interact with the Platform services implementation available on the nanosatellite, and can be monitored and controlled from ground by taking advantage of the CCSDS-standardized M&C services.

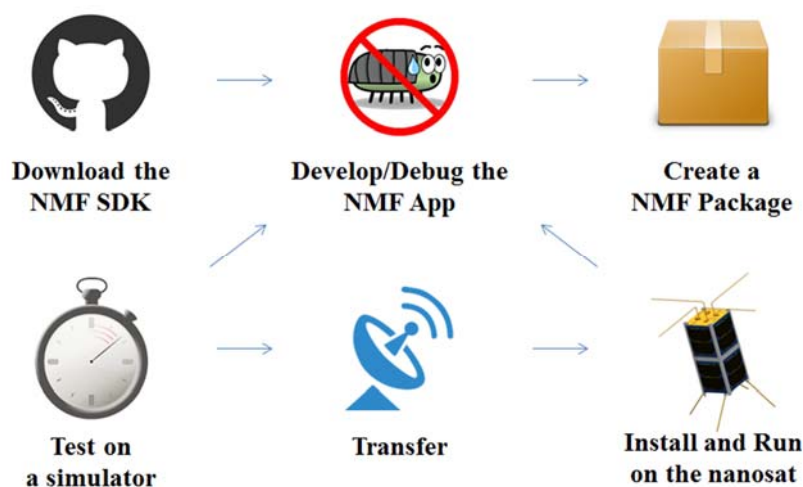


Figure 1: The lifecycle of an NMF App

An NMF app can be developed using the NMF SDK and then packaged into a NMF Package using the NMF Package Assembler. After, the app can be tested on a simulator which can be purely software-based or a Flatsat that emulates the nanosatellite's hardware and software. The NMF Package is then transferred to the nanosatellite where it can be installed and started at any time. If a problem is found during the execution in the nanosatellite, it is possible to go back to the development phase and repeat the process.

4 BUILD THE NANOSAT MO FRAMEWORK

NMF SDK is distributed both in a form of prebuilt binaries, and source code. In order to build it manually, please follow the instructions available under:

<https://github.com/esa/nanosat-mo-framework#building>

NMF Apps use the implementation of the NanoSat MO Connector in order to connect to the NanoSat MO Supervisor and register itself in the Central Directory service to become visible from ground. The NMF Apps source code examples from the SDK can now be built.

5 DEVELOP AN NMF APP

5.1 Creating a project

The software development platform NetBeans is suggested to be used for the development of NMF Apps. The SDK includes many NMF Apps source code examples in:
sdk/examples/space

The developer is advised to create a project inside that folder by duplicating the demo-hello-world project.

The following steps are recommended:

1. Go to: sdk/examples/space
2. Duplicate the folder: hello-world
3. Rename the folder of the new project to the desired name
4. Go to the new project folder and edit the following property in the provider.properties file: helpertools.configurations.OrganizationName
5. Open the newly created project in NetBeans
6. Refactor the DemoHelloWorld.java class to the desired name
7. Edit the following xml tags of the pom.xml file, inside the “Project Files”:
 - a. artifactId
 - b. version
 - c. name
 - d. description
 - e. url
 - f. organization tags
 - g. developers tags
 - h. assembly.mainClass
8. Clean and Build the project.

The project is now ready for development!

5.2 Initializing the NanoSat MO Connector

The initialization of the NanoSat MO Connector is extremely simple. The newly created project already contains an example of the NanoSat MO Connector implementation object instantiation:

```
private final NanoSatMOConnectorImpl connector = new NanoSatMOConnectorImpl();
```

To initialize the connector it is necessary to pass the M&C adapter that will be called for the monitor and control of parameters and actions. One can extend 1 of the following 2 adapters: the MonitorAndControlNMFAdapter or the SimpleMonitorAndControlAdapter. These adapters are described in more detail in section 5.3.1.



The project includes the initialization of the connector object with a MCAdapterSimple:

```
public DemoHelloWorld() {
    connector.init(new MCAdapterSimple());
}
```

Three files are read during the initialization process:

- provider.properties
- settings.properties
- transport.properties

The first file is expected to be configured by the developer and it includes configurations for the NMF App. The second and third file are expected to be static files that are configured by the mission where the NMF App is running. An example of the provider.properties is presented:

```
# MO App configurations
helpertools.configurations.OrganizationName=esa
helpertools.configurations.provider.app.category=NMF_App

# NanoSat MO Framework transport configuration
helpertools.configurations.provider.transportfilepath=../transport.properties

# NanoSat MO Framework Settings
esa.mo.nanosatmoframework.provider.settings=../settings.properties

# NanoSat MO Framework dynamic configurations
esa.mo.nanosatmoframework.provider.dynamicchanges=true

# Archive flag to drop the table
esa.mo.com.impl.provider.ArchiveManager.droptable=false
```

Section 8.3 includes more details on how to change the transport, however this is something that should be static and configured by the mission.

During initialization, the connector will start the services and write a file with the connection details for all the services. This includes the respective service URI, Broker URI, domain, and service key. The file can be found in the folder where the application is running and it is named “providerURIs.properties”.

5.3 Monitor and Control integration

This section covers the monitor and control functionalities provided by the NanoSat MO Connector, this includes the M&C adapters for parameters and actions, pushing parameter values, raising alerts and reporting the execution progress of actions.

There are 2 possible ways of receiving parameter values by a consumer:

1. On Request (and/or periodically)
2. Asynchronously pushing them to the consumer

The developer must choose one of the 2 ways (or both simultaneously) depending on the use case that fits best. The implementation will be different depending on the chosen way, for the first case, the M&C adapter must be implemented, for the second case, the developer will be “Pushing a Parameter Value” to the consumers.

A set of parameter values can be acquired together. This is called an “Aggregation” and they can periodically collect multiple parameter values and report them to the consumer.

5.3.1 M&C Adapters

One of two adapters can be extended for the development of customized monitoring and control adapter:

- MonitorAndControlNMFAdapter
- SimpleMonitorAndControlAdapter

By implementing one of these two adapters and then use them for the initialization of the NanoSat MO Connector, the NMF App becomes accessible for interactions with consumers. The DemoHelloWorld class already comes with an example for the implementation of both adapters.

MonitorAndControlNMFAdapter:

```
public class MCAdapter extends MonitorAndControlNMFAdapter {

    @Override
    public void initialRegistrations(MCRegistration registrationObject) {
        registrationObject.setMode(RegistrationMode.DONT_UPDATE_IF_EXISTS);

        // ----- Parameters -----
        final ParameterDefinitionDetailsList defs = new
ParameterDefinitionDetailsList();
        final IdentifierList names = new IdentifierList();

        defs.add(new ParameterDefinitionDetails(
            PARAMETER_DESCRIPTION,
            Union.STRING_SHORT_FORM.byteValue(),
            " ",
            false,
            new Duration(3),
            null,
            null
        ));
    }
}
```

```

        names.add(new Identifier(PARAMETER_NAME));
        registrationObject.registerParameters(names, defs);
    }

    @Override
    public Attribute onGetValue(Identifier identifier, Byte rawType) {
        if (PARAMETER_NAME.equals(identifier.getValue())) {
            return (Attribute) HelperAttributes.javaType2Attribute(var);
        }

        return null;
    }

    @Override
    public Boolean onSetValue(IdentifierList identifiers,
        ParameterRawValueList values) {
        if (PARAMETER_NAME.equals(identifiers.get(0).getValue())) {
            var = values.get(0).getRawValue().toString();
            return true; // to confirm that the variable was set
        }

        return false;
    }

    @Override
    public UIInteger actionArrived(Identifier name, AttributeValueList
        attributeValues,
        Long actionInstanceObjId, boolean reportProgress, MALInteraction
        interaction) {
        return null; // Action service not integrated
    }
}

```

The `initialRegistrations` method needs to be overridden in order to register the selected parameter definitions, aggregation definitions, alert definitions, and action definitions. These can be registered by using the registration object passed as argument.

Please notice that the overridden `onGetValue` method has a conversion occurring from a java primitive type (String) to a MAL data type (String encapsulated in a Union type). All Java primitive types need to be converted to MAL data types (Attribute types) therefore there are static methods already available: `javaType2Attribute` and `attribute2JavaType` implemented in the `HelperAttributes` class in order to perform this conversion.

Important: If the developer intends to use MAL Attribute data types directly, there is a very useful MAL data types table available in section 9.

SimpleMonitorAndControlAdapter:

The Simple M&C Adapter (`SimpleMonitorAndControlAdapter` class) is a simplified version of the `MonitorAndControlNMFAdapter` adapter already explained above. This adapter



completely abstracts the developer from the MAL data types and allows the direct use of Java primitive types.

The disadvantage of using this adapter is the loss of some functionality and the loss in the ability to express specific MAL data types, for example, an 'Identifier' type.

```
public class MCAdapterSimple extends SimpleMonitorAndControlAdapter {

    @Override
    public void initialRegistrations(MCRegistration registrationObject) {
        registrationObject.setMode(RegistrationMode.DONT_UPDATE_IF_EXISTS);

        // ----- Parameters -----
        final ParameterDefinitionDetailsList defs = new
ParameterDefinitionDetailsList();
        final IdentifierList names = new IdentifierList();

        defs.add(new ParameterDefinitionDetails(
            PARAMETER_DESCRIPTION,
            Union.STRING_SHORT_FORM.byteValue(),
            "",
            false,
            new Duration(3),
            null,
            null
        ));
        names.add(new Identifier(PARAMETER_NAME));
        registrationObject.registerParameters(names, defs);
    }

    @Override
    public Serializable onGetValueSimple(String name) {
        if (PARAMETER_NAME.equals(name)) {
            return var;
        }

        return null;
    }

    @Override
    public boolean onSetValueSimple(String name, Serializable value) {
        if (PARAMETER_NAME.equals(name)) {
            var = value.toString();
            return true; // to confirm that the variable was set
        }

        return false;
    }

    @Override
    public boolean actionArrivedSimple(String name, Serializable[] values,
        Long actionInstanceId) {
        return false;
    }
}
```

The names of the methods to be overridden are very similar with the difference that they include the word “Simple” in front of it. Also, the arguments of the methods are not MAL data types and have been simplified for a developer that doesn’t want to understand the MAL data types.

5.3.2 *Pushing a Parameter Value*

In order to push a parameter value from the NMF App to the consumers, one can use the `pushParameterValue` or `pushMultipleParameterValues` methods available in the NanoSat MO Connector class.

The first method will automatically take care of converting it to a MAL Element data type in case the submitted value is a java primitive data type (for example: int, float, etc). The method also supports serializable objects by automatically serializing it into a Blob Mal data type before pushing it.

An example on the usage of this method is presented in the “Push Clock” demo, where 4 parameters: day of the week, hours, minutes and seconds are pushed to the consumer. The example for pushing the seconds parameter follows:

```
if (seconds != calendar.get(Calendar.SECOND)) {
    seconds = calendar.get(Calendar.SECOND);
    connector.pushParameterValue("Seconds", seconds);
}
```

The Consumer Test Tool (CTT) can be used to verify that the push of parameter values is indeed occurring. The “Published Parameter Values” tab presents a visual representation of the parameter values ordered by their object instance identifier.

Please notice that in the “Parameter service” tab, the parameter definitions were automatically added during the “push” of the first parameter without having to register it.

5.3.3 *Raising an Alert*

In order to raise an alert from the NMF App to the consumer, one can use the `pushAlertEvent` method available in the NanoSat MO Connector class.

An example using this method can be found in the ‘10 seconds Alert’ demo:

```
try {
    connector.publishAlertEvent("10SecondsAlert", null);
} catch (NMFException ex) {
    Logger.getLogger(Demo10secAlert.class.getName()).log(Level.SEVERE,
        "The Alert could not be published to the consumer!", ex);
}
```



The demo raises an alert to the consumers every 10 seconds using a timer.

The method `publishAlertEvent` uses the CCSDS M&C Alert service to publish its alerts. It takes the name of the alert and a list of argument values as input. If the alert does not exist in the service, then it will be automatically created. The list of argument values can be null just as it is presented in the snip above.

It is important to know that the Alert service uses the Event service (from COM services) to publish its alerts. The Consumer Test Tool (CTT) can be used to check that the alerts are being raised by verifying the “Event service” tab.

5.3.4 Reporting Action Execution Progress

In order to report the execution progress of an action from the NMF App to the consumer, one can use the `reportActionExecutionProgress` method available in the NanoSat MO Connector class. The method is intended to be used after an action has been called by a consumer.

The “5 stages Action” demo presents an example of the method being used:

```
public void reportFiveStepsAction(Long actionId) throws NMFException {
    for (int stage = 1; stage < TOTAL_N_OF_STAGES + 1; stage++) {
        connector.reportActionExecutionProgress(true, 0, stage,
TOTAL_N_OF_STAGES, actionId);

        try { // Quick and dirty, but enough for demo purposes!
            Thread.sleep(SLEEP_TIME * 1000); // 1000 is ms multiplier.
        } catch (InterruptedException ex) {
            Thread.currentThread().interrupt();
        }
    }
}
```

The method `reportActionExecutionProgress` uses the CCSDS COM Activity Tracking service to report the execution progress of an action. It takes as input: a boolean for the success status, an integer for the error number (not used if the success is true), an integer for the progress stage, an integer for the total number of progress stages, and a long for the action instance identifier (value provided by the `actionArrived` method).

It is important to understand that the Activity Tracking service uses the Event service (from COM) to publish its events. The Consumer Test Tool (CTT) can be used to check that the action execution progress reports are being reported by verifying the “Event service” tab.

Important: There is a difference between “progress stages” and “execution stages”. The execution stages contain all the progress stages plus two additional stages: initial stage and final stage. These 2 are taken care automatically by the Action service, therefore the



developer only needs to report the “progress stages”. The following table shows the relation between the “progress stages” and “execution stages” for the “5 stages Action” demo:

| initial stage | progress stages | | | | | final stage |
|------------------|-----------------|---|---|---|---|-------------|
| | 1 | 2 | 3 | 4 | 5 | |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| execution stages | | | | | | |

Please notice that if one checks the Execution COM object on the Event service, the object body will only hold the “execution stage” of the action and not the reported “progress stage” that was used on the reportActionExecutionProgress method.

5.4 Accessing the Platform services

All the Platform services can be accessed from NanoSat MO Connector from the `getPlatformServices` method:

```
try {
    PlatformServicesConsumer platform = connector.getPlatformServices();
} catch (NMFException ex) {
    Logger.getLogger(DemoApp.class.getName()).log(Level.SEVERE,
        "The Platform services are not available.", ex);
}
```

The `PlatformServicesConsumer` class includes getters to all the respective platform services.

The developer is strongly encouraged to go through the Platform services documentation which is available in the SDK output under:

`docs/MO_services/ServiceSpecPlatform.pdf`

In the next sections, each service utilization will be presented in more detail.

5.4.1 Camera

The Camera service allows a consumer to acquire pictures and control a camera in the spacecraft platform. The service can perform format conversions in case the consumer selects a specific format other than raw. The service can also stream pictures periodically.

The Camera service can be reached from the Camera service stub:

```
try {
    PlatformServicesConsumer platform = connector.getPlatformServices();
    CameraStub stub = platform.getCameraService();
} catch (NMFException ex) {
    Logger.getLogger(DemoApp.class.getName()).log(Level.SEVERE,
        "The Platform services are not available.", ex);
} catch (IOException ex) {
    Logger.getLogger(DemoApp.class.getName()).log(Level.SEVERE,
        "The service is not available.", ex);
}
```

In order to take a picture, it is necessary to use the `takePicture` method. An example using this method can be found in the ‘SnapNMF’ demo:

```
connector.getPlatformServices().getCameraService().takePicture(
    resolution,
    PictureFormat.RAW,
    new Duration(0.200),
    new DataReceivedAdapter(actionInstanceId)
);
```



The DataReceiverAdapter adapter implements the CameraAdapter class. This adapter will be called upon receiving the picture from the Camera service.

5.4.2 GPS

The GPS service provides the ability to retrieve satellite navigation data from a Global Navigation Satellite System (GNSS) device receiver in the spacecraft platform. The GPS service provides the capability for streaming NMEA messages; the capability for enabling/disabling the streaming of NMEA messages; the capability for getting the last known position from the receiver; the capability for getting the satellites GNSS information; the capability for maintaining the list of nearby position events.

The nearbyPosition operation allows a consumer to receive a message from the service when the spacecraft enters or exists a certain position. These can be set using the addNearbyPosition and removed using the removeNearbyPosition.

The GPS service can be reached from the GPS service stub:

```
try {
    PlatformServicesConsumer platform = connector.getPlatformServices();
    GPSSub stub = platform.getGPSService();
} catch (NMFException ex) {
    Logger.getLogger(DemoApp.class.getName()).log(Level.SEVERE,
        "The Platform services are not available.", ex);
} catch (IOException ex) {
    Logger.getLogger(DemoApp.class.getName()).log(Level.SEVERE,
        "The service is not available.", ex);
}
```

In order to get the last known position from the GPS unit, it is necessary to use the getLastKnownPosition method. An example using this method can be found in the 'GPSData' demo:

```
try {
    GetLastKnownPositionResponse pos =
connector.getPlatformServices().getGPSService().getLastKnownPosition();

    if (PARAMETER_GPS_LATITUDE.equals(identifier.getValue())) {
        return (Attribute)
HelperAttributes.javaType2Attribute(pos.getBodyElement0().getLatitude());
    }

    if (PARAMETER_GPS_LONGITUDE.equals(identifier.getValue())) {
        return (Attribute)
HelperAttributes.javaType2Attribute(pos.getBodyElement0().getLongitude());
    }

    if (PARAMETER_GPS_ALTITUDE.equals(identifier.getValue())) {
```

```

        return (Attribute)
HelperAttributes.javaType2Attribute(pos.getBodyElement0().getAltitude());
    }
    } catch (IOException ex) {
        Logger.getLogger(DemoGPSData.class.getName()).log(Level.SEVERE,
null, ex);
    } catch (NMFException ex) {
        Logger.getLogger(DemoGPSData.class.getName()).log(Level.SEVERE,
null, ex);
    }
}

```

5.4.3 AutonomousADCS

The AutonomousADCS service allows a consumer to monitor the attitude from an ADCS device in the spacecraft platform and to set/unset the desired attitude from a list of attitude definitions.

The AutonomousADCS service can be reached from the AutonomousADCS service stub:

```

try {
    PlatformServicesConsumer platform = connector.getPlatformServices();
    AutonomousADCSSub stub = platform.getAutonomousADCSService();
} catch (NMFException ex) {
    Logger.getLogger(DemoApp.class.getName()).log(Level.SEVERE,
        "The Platform services are not available.", ex);
} catch (IOException ex) {
    Logger.getLogger(DemoApp.class.getName()).log(Level.SEVERE,
        "The service is not available.", ex);
}

```

It is necessary to add an Attitude Definition before setting the desired attitude. This can be achieved by using the addAttitudeDefinition method. An example can be found in the 'AllInOne' demo:

```

try {
    IdentifierList names = new IdentifierList();
    names.add(sunDef.getName());
    names.add(nadirDef.getName());
    LongList objIds =
nmf.getPlatformServices().getAutonomousADCSService().listAttitudeDefinition(names);

    sunPointingObjId = objIds.get(0);
    nadirPointingObjId = objIds.get(1);

    if (sunPointingObjId == null) { // It does not exist
        LongList sunObj =
nmf.getPlatformServices().getAutonomousADCSService().addAttitudeDefinition(sunDe
fs);
        sunPointingObjId = sunObj.get(0);
    }
}

```



```

        if (nadirPointingObjId == null) { // It does not exist
            LongList nadirObj =
nmf.getPlatformServices().getAutonomousADCSService().addAttitudeDefinition(nadir
Defs);
            nadirPointingObjId = nadirObj.get(0);
        }
    } catch (IOException ex) {
        Logger.getLogger(MCAllInOneAdapter.class.getName()).log(Level.SEVERE,
null, ex);
    } catch (MALInteractionException ex) {
        if
(ex.getStandardError().getErrorNumber().equals(COMHelper.DUPLICATE_ERROR_NUMBER)
) {
            Logger.getLogger(MCAllInOneAdapter.class.getName()).log(Level.INFO,
"The Attitude Definition already exists!");
        } else {

Logger.getLogger(MCAllInOneAdapter.class.getName()).log(Level.SEVERE, null, ex);
        }
    } catch (MALErrorException ex) {
        Logger.getLogger(MCAllInOneAdapter.class.getName()).log(Level.SEVERE,
null, ex);
    } catch (NMFException ex) {
        Logger.getLogger(MCAllInOneAdapter.class.getName()).log(Level.SEVERE,
null, ex);
    }
}

```

The `setDesiredAttitude` method allows the selection of the desired attitude based the object instance identifier of an attitude definition. An example can be found in the 'AllInOne' demo:

```

try {
    System.out.println(ACTION_SUN_POINTING_MODE + " with value is ["
        + HelperAttributes.attribute2string(argValue) + "]);

    nmf.getPlatformServices().getAutonomousADCSService().setDesiredAttitude(
        sunPointingObjId,
        (Duration) argValue,
        new Duration(2)
    );
} catch (MALInteractionException ex) {
    Logger.getLogger(MCAllInOneAdapter.class.getName()).log(Level.SEVERE,
null, ex);
    return new UIInteger(3);
} catch (IOException ex) {
    Logger.getLogger(MCAllInOneAdapter.class.getName()).log(Level.SEVERE,
null, ex);
} catch (MALErrorException ex) {
    Logger.getLogger(MCAllInOneAdapter.class.getName()).log(Level.SEVERE,
null, ex);
    return new UIInteger(3);
} catch (NMFException ex) {

```



```

        Logger.getLogger(MCAllInOneAdapter.class.getName()).log(Level.SEVERE,
null, ex);
        return new UInteger(3);
    }

```

In order to monitor the attitude, it is necessary to use the register for the monitorAttitude operation. An example can be found in the 'AllInOne' demo:

```

try {
    // Subscribe monitorAttitude

nmf.getPlatformServices().getAutonomousADCSService().monitorAttitudeRegister(
    ConnectionConsumer.subscriptionWildcard(),
    new DataReceivedAdapter()
);
} catch (IOException ex) {
    Logger.getLogger(MCAllInOneAdapter.class.getName()).log(Level.SEVERE,
null, ex);
} catch (MALInteractionException ex) {
    Logger.getLogger(MCAllInOneAdapter.class.getName()).log(Level.SEVERE,
null, ex);
} catch (MALErrorException ex) {
    Logger.getLogger(MCAllInOneAdapter.class.getName()).log(Level.SEVERE,
null, ex);
} catch (NMFException ex) {
    Logger.getLogger(MCAllInOneAdapter.class.getName()).log(Level.SEVERE,
null, ex);
}

```

5.4.4 *Software-defined Radio*

The Software-defined Radio provides a generic mechanism to set, configure and receive data from a Software-defined Radio device.

The Software-defined Radio service can be reached from the Software-defined Radio service stub:

```

try {
    PlatformServicesConsumer platform = connector.getPlatformServices();
    SoftwareDefinedRadioStub stub =
platform.getSoftwareDefinedRadioService();
} catch (NMFException ex) {
    Logger.getLogger(DemoApp.class.getName()).log(Level.SEVERE,
"The Platform services are not available.", ex);
} catch (IOException ex) {
    Logger.getLogger(DemoApp.class.getName()).log(Level.SEVERE,
"The service is not available.", ex);
}

```

5.4.5 *Optical Data Receiver*

The Optical Data Receiver service provides a mechanism to receive messages from an Optical Data Receiver device.

The Optical Data Receiver service can be reached from the Optical Data Receiver service stub:

```
try {
    PlatformServicesConsumer platform = connector.getPlatformServices();
    OpticalDataReceiverStub stub = platform.getOpticalDataReceiverService();
} catch (NMFException ex) {
    Logger.getLogger(DemoApp.class.getName()).log(Level.SEVERE,
        "The Platform services are not available.", ex);
} catch (IOException ex) {
    Logger.getLogger(DemoApp.class.getName()).log(Level.SEVERE,
        "The service is not available.", ex);
}
```

5.4.6 *Magnetometer*

The Magnetometer service provides a generic mechanism to retrieve the magnetic field from a magnetometer in the spacecraft platform.

The Magnetometer service can be reached from the Magnetometer service stub:

```
try {
    PlatformServicesConsumer platform = connector.getPlatformServices();
    MagnetometerStub stub = platform.getMagnetometerService();
} catch (NMFException ex) {
    Logger.getLogger(DemoApp.class.getName()).log(Level.SEVERE,
        "The Platform services are not available.", ex);
} catch (IOException ex) {
    Logger.getLogger(DemoApp.class.getName()).log(Level.SEVERE,
        "The service is not available.", ex);
}
```

5.4.7 *Power Control*

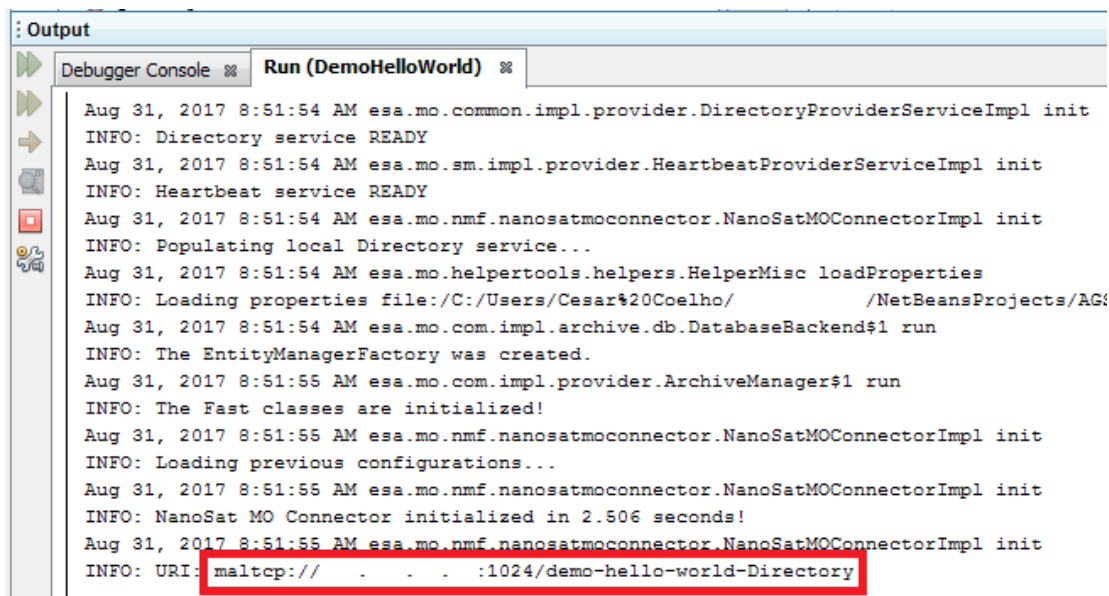
The Power Control service provides a generic mechanism to list the available power units in a spacecraft platform and to enable/disable them.

The Power Control service can be reached from the Power Control service stub:

```
try {
    PlatformServicesConsumer platform = connector.getPlatformServices();
    PowerControlStub stub = platform.getPowerControlService();
} catch (NMFException ex) {
    Logger.getLogger(DemoApp.class.getName()).log(Level.SEVERE,
        "The Platform services are not available.", ex);
} catch (IOException ex) {
    Logger.getLogger(DemoApp.class.getName()).log(Level.SEVERE,
        "The service is not available.", ex);
}
```

6 TESTING THE APP

After building the NMF App, one can run it!



```

: Output
Debugger Console  Run (DemoHelloWorld)
Aug 31, 2017 8:51:54 AM esa.mo.common.impl.provider.DirectoryProviderServiceImpl init
INFO: Directory service READY
Aug 31, 2017 8:51:54 AM esa.mo.sm.impl.provider.HeartbeatProviderServiceImpl init
INFO: Heartbeat service READY
Aug 31, 2017 8:51:54 AM esa.mo.nmf.nanosatmoconnector.NanoSatMOConnectorImpl init
INFO: Populating local Directory service...
Aug 31, 2017 8:51:54 AM esa.mo.helpertools.helpers.HelperMisc loadProperties
INFO: Loading properties file:/C:/Users/Cesar%20Coelho/ /NetBeansProjects/AG!
Aug 31, 2017 8:51:54 AM esa.mo.com.impl.archive.db.DatabaseBackend$1 run
INFO: The EntityManagerFactory was created.
Aug 31, 2017 8:51:55 AM esa.mo.com.impl.provider.ArchiveManager$1 run
INFO: The Fast classes are initialized!
Aug 31, 2017 8:51:55 AM esa.mo.nmf.nanosatmoconnector.NanoSatMOConnectorImpl init
INFO: Loading previous configurations...
Aug 31, 2017 8:51:55 AM esa.mo.nmf.nanosatmoconnector.NanoSatMOConnectorImpl init
INFO: NanoSat MO Connector initialized in 2.506 seconds!
Aug 31, 2017 8:51:55 AM esa.mo.nmf.nanosatmoconnector.NanoSatMOConnectorImpl init
INFO: URI: maltcp:// :1024/demo-hello-world-Directory
  
```

The Consumer Test Tool (CTT) can be used to test some of the operations of the NMF App and also to observe its correct behaviour during execution. The CTT is available under:
`<sdk-target-dir>\bin\tools\consumer-test-tool`

One use the Directory service URI to reach the NMF App. This URI is printed by the NMF App after initialization as presented in the figure above. It can also be obtained from the automatically generated `providerURIs.properties` file available in the folder where the application is running.

After copying the URI, go to the “Communication Settings” tab and insert it into the Directory service URI textbox. After, press “Fetch Information” followed by “Connect to selected provider”.

Info: CTT source code project is also available in the src folder of the SDK and can be easily extended by the experimenters if they wish to do so.

6.1 Run the NMF App in the SDK environment

The instructions are available under:

<https://github.com/esa/nanosat-mo-framework/tree/master/sdk#runningdebugging-the-applications-from-the-ide>

6.2 Automatic deployment of the NMF App

The instructions are available under:

<https://github.com/esa/nanosat-mo-framework/tree/master/sdk#adding-the-application-to-the-sdk-packaging>

6.3 Project with extra external dependencies

It is common to develop a project having external dependencies besides the NanoSat MO Framework. To compile and make the app executable in the sandbox folder, the developer will need to build his project with maven assembly plugin.

1. Open the pom.xml file of the project
2. Uncomment the “maven-assembly-plugin” in the plugins area
3. Add the “<scope>provided</scope>” tag in the NANOSAT_MO_CONNECTOR dependency. This is necessary to avoid including the framework in the assembled jar.
4. Continue with the steps on the section above. Please notice that in the target folder there will be 2 jars: one containing the dependencies (the filename ending is “-jar-with-dependencies”) and the other without dependencies. The file to be copied to the sandbox shall be the jar with dependencies.

PROTIP: The executable files must link to the correct jar file with dependencies.



7 HANDS-ON ACTIVITIES

7.1 Activity 1

Create a project named: Activity1

Write a small app that outputs the string “This is my first app!” when the parameter “myFirstParameter” is called.

Execute the app in the SDK environment and use CTT to try the NMF App.

7.2 Activity 2

Create a project named: Activity2

Write a small app that reports 10 execution stages after the action “myFirstAction” is called. Additionally, the app must raise an alert between the 4th and 5th execution stages. Execute the NMF App in the SDK environment and use CTT to try it.

Hint: Use the Event service tab to check if the reporting of Actions were successful and if the Alert was raised between the 4th and 5th execution stages report.

8 FAQ

8.1 How to convert from a Java primitive data type to a MAL data type and vice versa?

The “MO Helper Tools” comes with the NanoSat MO Framework and it is a toolbox that facilitates many of the common functionalities needed during the development of MO-related software.

In the class `esa.mo.helptools.helpers.HelperAttributes` there are two methods:

To convert from Java primitive data type to a MAL data type:

```
public static Object javaType2Attribute(Object obj);
```

To convert from MAL data type to a Java primitive data type:

```
public static Object attribute2JavaType(Object obj);
```

8.2 How to drop the COM Archive database table at start up?

In order to drop the COM Archive database at start up, one must switch to true the following property in the `provider.properties` file:

```
# Archive flag to drop the table
esa.mo.com.impl.provider.ArchiveManager.droptable=true
```

8.3 How to change the transport layer?

Add a `transport.properties` file in the NMF App folder with the desired transport. Then, change the `provider.properties` file to point to this new file:

```
# NanoSat MO Framework transport configuration
helptools.configurations.provider.transportfilepath=transport.properties
```

The `transport.properties` allows the selection of the desired transport by changing the hash characters at the beginning of the file. The transport without the hash is the selected one:

```
# The following sets the default protocol used
#org.ccsds.moims.mo.mal.transport.default.protocol = malhttp://
#org.ccsds.moims.mo.mal.transport.default.protocol = rmi://
org.ccsds.moims.mo.mal.transport.default.protocol = maltcp://
```

Additionally, a secondary transport can be set with the property:

```
# The following sets the secondary protocol used
#org.ccsds.moims.mo.mal.transport.secondary.protocol = rmi://
```

9 MAL ATTRIBUTE DATA TYPES

Please notice that there are static methods already available: `javaType2Attribute` and `attribute2JavaType` in the `esa.mo.helpertools.helpers.HelperAttributes` class in order to convert from and to Java primitive data types. However, if the developer intends to use the MAL-specific data types, this section will aid that process.

To create a new MAL Attribute data type, the process is straightforward for most of the Attributes, for example:

```
new Identifier("TheIdentifierString");
```

Please notice that the MAL Attributes containing an already existing name like the java primitive type, need an extra encapsulation in order to become MAL data types. The “Union” type must be used for: Boolean, Integer, Long, String, Double, Float, Byte, Short. These are marked with a red asterisk * in the table.

Java primitive Integer type must be wrapped into a MAL Union type:

```
new Union((Integer) obj);
```

| MAL Attributes | | |
|-----------------|------------------------|---|
| <u>Name</u> | <u>Short Form Part</u> | <u>Description</u> |
| Blob | 1 | The Blob structure is used to store binary object attributes. It is a variable-length, unbounded, octet array. The distinction between this type and a list of Octet attributes is that this type may allow language mappings and encodings to use more efficient or appropriate representations. |
| Boolean* | 2 | The Boolean structure is used to store Boolean attributes. Possible values are ‘True’ or ‘False’. |
| Duration | 3 | The Duration structure is used to store Duration attributes. It represents a length of time in seconds. It may contain a fractional component. |
| Float* | 4 | The Float structure is used to store floating point attributes using the IEEE 754 32-bit range. Three special values exist for this type: POSITIVE_INFINITY, NEGATIVE_INFINITY, and NaN (Not A Number). |
| Double* | 5 | The Double structure is used to store floating point attributes |

| | | |
|-------------------|----|---|
| | | using the IEEE 754 64-bit range. Three special values exist for this type: POSITIVE_INFINITY, NEGATIVE_INFINITY, and NaN (Not A Number). |
| Identifier | 6 | The Identifier structure is used to store an identifier and can be used for indexing. It is a variable-length, unbounded, Unicode string. |
| Octet | 7 | The Octet structure is used to store 8-bit signed attributes. The permitted range is -128 to 127. |
| UOctet | 8 | The UOctet structure is used to store 8-bit unsigned attributes. The permitted range is 0 to 255. |
| Short* | 9 | The Short structure is used to store 16-bit signed attributes. The permitted range is -32768 to 32767. |
| UShort | 10 | The UShort structure is used to store 16-bit unsigned attributes. The permitted range is 0 to 65535. |
| Integer* | 11 | The Integer structure is used to store 32-bit signed attributes. The permitted range is -2147483648 to 2147483647. |
| UInteger | 12 | The UInteger structure is used to store 32-bit unsigned attributes. The permitted range is 0 to 4294967295. |
| Long* | 13 | The Long structure is used to store 64-bit signed attributes. The permitted range is -9223372036854775808 to 9223372036854775807. |
| ULong | 14 | The ULong structure is used to store 64-bit unsigned attributes. The permitted range is 0 to 18446744073709551615. |
| String* | 15 | The String structure is used to store String attributes. It is a variable-length, unbounded, Unicode string. |
| Time | 16 | The Time structure is used to store absolute time attributes. It represents an absolute date and time to millisecond resolution. |
| FineTime | 17 | The FineTime structure is used to store high-resolution absolute time attributes. It represents an absolute date and time to picosecond resolution. |
| URI | 18 | The URI structure is used to store URI addresses. It is a variable-length, unbounded, Unicode string. |