OpenIO Labs - ScriptML

LWM2M Support in ScriptML

UG404 (v2017.1) April 3, 2017

Revision History

Date	Version	Revisions
3/4/2017	2017.1	*Initial release.*

Table of Contents

Chapter 1: Introduction

This User Guide is intended to provide a brief overview of the LWM2M support in the OpenIO Labs system, a discussion on the different objects supported, and a guide into the use of these in the OpenIO Labs system using ScriptML.

The User Guide assumes that you are familiar with the OpenIO Labs system, ScriptML and accessing the system via a web browser. For those not familiar with these, refer to References [1,2,5] as outlined in the References section of this guide. In addition it is recommended that you familiarise yourself with the LWM2M protocol used in the OpenIO Labs system, and also the JSON object serialisation language. Details for both of these can be found in Ref [3].

The structure of the OpenIO Labs system is covered in detail in Ref [1] and will only be summarised here. From the perspective of this User Guide, the system comprises edge devices (IOIs) connected to a server, the server with a range of databases and a browser that access the server and databases. The user interacts with the OpenIO Labs system (including the server and the IOIs) by way of the browser interface.

The IOIs and the OpenIO Labs server communicate using the industry standard LWM2M protocol (See Ref [3] for further details). The LWM2M protocol is responsible for managing the communications between the IOI and the server. These communications include Registration, Data Transfer, Control and Notification messages.

One of the key components of the LWM2M protocol is the definition of Objects and Resources it uses. The client and server communicate using these Objects and Resources to exchange data. The Objects can be considered as the "Thing" we wish to exchange between client and server, and the Resources are the details and attributes for this "Thing".

In this User Guide, we will go into some detail of what the Objects and Resources are, how they are represented in the OpenIO Labs system, describe some OpenIO Labs specific Objects and finally describe how the user can use ScriptML to manipulate and use the LWM2M Objects and Resources. To this end, some simple examples of accessing LWM2M Objects using C and Python scripts are presented.

Chapter 2: Objects and Resources

Introduction to Client and ScriptML LWM2M Support

The LWM2M specification defines a protocol which regulates the interactions between a client and a server. The details of the specification are beyond the scope of this document, but are fully defined in the LWM2M Technical Specification (Ref [3]).

The OpenIO Labs system implements both the client and the server end of these exchange, and uses the LWM2M protocol to support the LWM2M exchanges.

The OpenIO Labs client further divides the structure for the use of the LWM2M protocol into two parts. These two parts are the client specific functions and the second the ScriptML functions.

The client functions are responsible for managing the relationship between the client and the server using the LWM2M protocol (Registration, De-registration, Read, Write, Observe etc.). These functions are managed in such a way that the end user (ScriptML script writer) is not troubled by the details for these aspects of the LWM2M protocol.

The ScriptML part is what the user will see and interact with. The client exposes a number of simple APIs that the user will use to create, manage and control objects and resources attached to the client (or device to use the LWM2M term).

By structuring the system in this way, the end user is presented with a very simple programming environment in which they can choose from a range of languages (such as C, Python, C++ etc.) to create complex experimental systems. The user will focus on the logic of the application that they wish to develop, but not be troubled with the complexity of interacting with the server using LWM2M.

LWM2M Objects and Resources

LWM2M is based on the concept of Objects and Resources. The Objects are the things that we may want to measure or control, and the Resources can be thought of as the attributes of the Object that define the details of the Object.

We will see shortly that a large number of Objects have been defined for use within LWM2M, and in addition, the OpenIO Labs system also defines a set of additional Objects that can be used.

Within LWM2M an Object can be a range of things, from a physical device, to the LWM2M security object, to a temperature measurement object. Pretty much anything that we need to interact with is defined as an object.

If we consider an example, such as a temperature measurement object. The object has a defined identification number (Object Id), which in the case of the temperature object is 3303. This temperature object has a number of resources associated with it. Similar to the Object Identifier, each resource also has an identifier to allow the client and server to exchange data relating to that specific object. For ease of understanding, the resources that the Temperature sensor may support are presented in the following table.

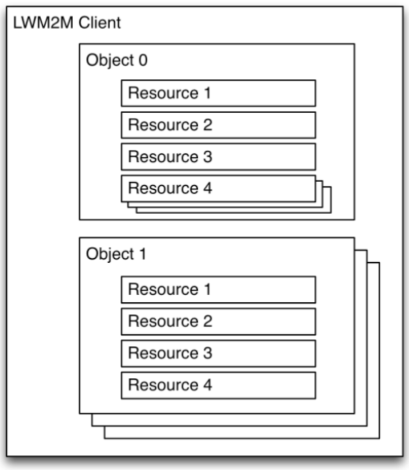
Resource	ID	Description
Sensor Value	5700	Last or Current Measured Value from the Sensor
Sensor Units	5701	Measurement Units Definition e.g. 'Cel' for Temperature in Celsius.
Min Measured Value	5601	The minimum value measured by the sensor since power ON or reset
Max Measured Value	5602	The maximum value measured by the sensor since power ON or reset
Min Range Value	5603	The minimum value that can be measured by the sensor
Max Range Value	5604	The maximum value that can be measured by the sensor
Reset Min Max to Current	5605	Reset the Min and Max Measured Values to Current Value

Resources Used by the Temperature Object (Object ID=3303)

In addition to the information presented in the table above, each of the resources has a number of associated parameters such as whether it is Read only, Write only, or Read and Write. The data type is also defined for the resource (such as Integer, Float, String, Boolean etc.). The number of instances of the resource associated with a specific object are defined, and finally, whether a specific resource is mandatory or optional for that specific object.

It should be noted that the Resource ID is unique for a resource type, but that resource type could be used in a range of objects. So, for example, Resource ID 5700 for the sensor value, will also be used in sensors such as Barometers, Magnetometers and Ammeters, amongst many others.

So within the client we can have multiple objects of multiple types. Each object will contain multiple resources of multiple types. This is illustrated in the figure below taken from the LWM2M specification.



The Internet Protocol for Smart Objects (IPSO) have defined a number of Objects and Resources that may be used with an LWM2M system. An example set of these Objects and Resources can be found in Ref [4].

LWM2M Objects Supported in the OpenIO Labs System

The OpenIO Labs System supports most of the objects defined by IPSO, Objects defined within the LWM2M standard and a number of Objects defined by OpenIO Labs.

The following table lists out the Objects that the system supports, their Object Identifiers and a brief description of how the object can be used.

The Objects are quite diverse in their use and operation, and not all objects will necessarily be required.

Object	ID	Object Description
IPSO Digital Input	3200	Generic digital input for non-specific sensors
IPSO Digital Output	3201	Generic digital output for non-specific actuators
IPSO Analog Input	3202	Generic analog input for non-specific sensors
IPSO Analog Output	3203	Generic object that can be used with any kind of analog output interface
IPSO Generic Sensor	3300	Used to report measurements from a generic sensor with sensor type and units included in the resource
IPSO Illuminance	3301	Used to report luminance levels from a light sensor
IPSO Presence	3302	Used to report presence from a presence sensor

IPSO Temperature	3303	Used to report temperature measurements from a temperature sensor
IPSO Humidity	3304	Used with a humidity sensor to report a humidity measurement
IPSO Power Measurement	3305	Used to report a power measurement from a power sensor
IPSO Actuation	3306	Used to actuate a device e.g. set ON/OFF
IPSO Set Point	3308	Used to fix a set point for e.g. a thermostat
IPSO Load Control	3310	Used for demand-response load control and other load control in automation application
IPSO Light Control	3311	Used to control a light source, such as an LED or other light
IPSO Power Control	3312	Used to control a power source, such as a Smart Plug
IPSO Accelerometer	3313	Used to represent a 1-3 axis accelerometer
IPSO Magnetometer	3314	Used to represent a 1-3 axis magnetometer with optional compass direction
IPSO Barometer	3315	Used with an air pressure sensor to report a barometer measurement
IPSO Voltage	3316	Used with voltmeter sensor to report measured voltage between two points
IPSO Ammeter	3317	Used with an ammeter to report measured electric current in amperes
IPSO Frequency	3318	Used to report frequency measurements
IPSO Depth	3319	Used to report the results of depth measurements
IPSO Percentage	3320	Used to report measurements as a percentage
IPSO Altitude	3321	Used to report altitude from an altitude sensor
IPSO Load	3322	Used to report the load (e.g. Force) on a load sensor
IPSO Pressure	3323	Used to report pressure measurements from a pressure sensor
IPSO Loudness	3324	Used to report loudness / noise levels
OIOL Script Management	7010	Used to transfer and monitor ScriptML script status
OIOL Camera Interface	7011	Provides an interface for OpenIO devices to send still images to the server

OIOL I2C Interface	7012	Provides an interface for I2C connected devices
OIOL SPI Interface	7013	Provides an interface for SPI connected devices
OIOL UART Interface	7014	Provides an interface for UART connected devices
OIOL Surface Plotter	7015	Used to push 3D surface data to the server
OIOL Remote Tagging Interface	7016	Allows remote devices to set-up tags for the data they provide
OIOL Motor Control	7017	Used to control a motor
OIOL Device Identifier	7018	Provides a device serial number for authentication with the server
OIOL Client Push Text	7019	Used to push a text object to the server

Summary of Objects Supported by OpenIO Labs System and ScriptML

LWM2M Data Transfer Formats and JSON

In addition to the Objects and Resources that are defined for the protocol, the LWM2M standard also defines data transfer formats that are used to transport the Objects and Resources. These data transfer formats define the data structures that will be used to encode objects, resources and the fields within both.

The data transfer formats that the LWM2M standard defines are optional and include plain text, opaque data, TLV formatted data and JSON formatted data.

Within the OpenIO Labs system we currently support the JSON data transfer format.

LWM2M JSON Formatting

Although the OpenIO Labs system uses JSON to transfer Objects and Resources between the client and the server, much of the formatting of the JSON strings is managed within the client domain of the client application. There are some instances, however, where, the end-user will need to create the JSON strings from the ScriptML domain and which will be embedded within other JSON objects within the client domain.

So, before moving deeper into the use of LWM2M, Objects and Resources, it is worth taking a brief aside to outline how JSON is used to format the Objects and Resources used in LWM2M. Later we will see some examples of the use of JSON formatting of objects, so we can consider this as an introduction to that section.

JSON is a serialisation protocol that uses the KEY:VALUE paradigm to encode data in the form of a string. JSON supports hierarchy, so we can have objects within objects, sequences of objects etc. etc.

A simple example of a JSON object, is one that OpenIO Labs Client Push Text Object uses (Object ID 7019).

The structure of the JSON object that captures the Text object is as follows:

```
 {
"objectName": "ObjectText",
"objectText": "Hello ScriptML",
"tags": ["my_text_tag"],
"pushToUI": true

}
```

It is worth explaining what each of these fields represents.

```
ObjectName:    This is the name of the Client Pushed Text Object
objectText:    This is the text that we wish to push to the server
tags:          The tags are used in the OpenIO Labs system to allow searching and
               collation of information
pushToUI:      This defines whether we want the text to be visible via the browser as it
               is pushed to the server
```

We can see that the basic structure of a JSON object is enclosed in braces ({}) and that the KEY is encapsulated by quotes and the value is represented either by a string (encapsulated in quotes) or by a JSON defined type (such as the boolean true in the example above).

Having considered the basic structure and use of the LWM2M Objects and Resources, we can now move on to consider how they can be used from within ScriptML. In this user guide we will consider both a C source representation as well as a Python source representation.

Chapter 3: LWM2M Support Using C

The OpenIO Labs support for LWM2M Objects and Resources within ScriptML using a C source file, is achieved through the use of the lwm2m.h header file. The header file is auto-generated within the OpenIO Labs build process from a JSON source file. The latest released version of the header file can be found in the include directory of the OpenIO Labs GIT repository.

When developing applications using ScriptML from a C source file you will not need to directly include the lwm2m.h header file as this will be pulled in by the server when translating the C source into the ScriptML executable. The presence of the lwm2m.h header file in the GIT repository is mainly to allow the end user to identify the names of variables and structures that the user will use when creating an application that uses the LWM2M Objects and Resources. In this section we will go through some examples of the use of the lwm2m.h header file when creating Objects and Resources for an application written in C.

In addition to the Objects and Resources defined in the header file, there are also C specific methods that are used to allow the ScriptML application to interact with the server. In this section we will go through the structure and use of these methods, also.

Accessing LWM2M Objects and Resources in C

The current version (as of writing) of the lwm2m.h header file is in excess of 6,000 lines long (a large percentage of which are comments). Given the length of the header file it is not realistic to reproduce it in its entirety here. Instead we will take a few examples and look through the structure of the header file.

IPSO Temperature Object – ID 3303

We will start by examining a reasonably representative LWM2M Object, namely the IPSO Temperature Object. If we review the table that we saw earlier for the temperature object, we have the following resources.

Resource	ID	Description
Sensor Value	5700	Last or Current Measured Value from the Sensor
Sensor Units	5701	Measurement Units Definition e.g. 'Cel' for Temperature in Celsius.
Min Measured Value	5601	The minimum value measured by the sensor since power ON or reset
Max Measured Value	5602	The maximum value measured by the sensor since power ON or reset
Min Range Value	5603	The minimum value that can be measured by the sensor
Max Range Value	5604	The maximum value that can be measured by the sensor
Reset Min Max to Current	5605	Reset the Min and Max Measured Values to Current Value

Resources Used by the Temperature Object (Object ID=3303)

Next if we explore the entry for the temperature object within the lwm2m.h header file. The following is the code snippet for the temperature object.

```
/**
 *  @brief IPSO Temperature
 *
 *  Not mandatory
 *
 *  Instance type: multiple
 *
 *  Description: This IPSO object should be used with a temperature sensor
 *  to report a temperature measurement.  It also provides resources for
 *  minimum/maximum measured values and the minimum/maximum range that can
 *  be measured by the temperature sensor. An example measurement unit is
 *  degrees Celsius (ucum:Cel).
 */
typedef struct
{
    /**
     *  @brief Sensor Value
     *
     *  Operations: read
     *
     *  Mandatory
     *
     *  Instance type: single
     *
     *  Units: defined by 'units' resource.
     *
     *  Type: float
     *
     *  Last or Current Measured Value from the Sensor
     */
    float LWM2M_RESOURCE( 5700, Sensor_Value );
    /**
     *  @brief Sensor Units
     *
     *  Operations: read
     *
     *  Not mandatory
     *
     *  Instance type: single
     *
     *  Type: string
     *
     *  Measurement Units Definition e.g. 'Cel' for Temperature in Celsius.
     */
    char * LWM2M_RESOURCE( 5701, Sensor_Units );
    /**
     *  @brief Reset Min and Max Measured Values
     *
     *  Operations: execute
     *
     *  Not mandatory
     *
     *  Instance type: single
     *
     *  Type: string
     *
     *  Reset the Min and Max Measured Values to Current Value
     */
    char * LWM2M_RESOURCE( 5605, Reset_Min_and_Max_Measured_Values );
    /**
     *  @brief Max Range Value
     *
     *  Operations: read
     *
     *  Not mandatory
     *
     *  Instance type: single
     *
     *  Units: defined by 'units' resource.
     *
     *  Type: float
     *
     *  The maximum value that can be measured by the sensor
     */
    float LWM2M_RESOURCE( 5604, Max_Range_Value );
    /**
     *  @brief Min Measured Value
     *
```

```
     *  Operations: read
     *
     *  Not mandatory
     *
     *  Instance type: single
     *
     *  Units: defined by 'units' resource.
     *
     *  Type: float
     *
     *  The minimum value measured by the sensor since power ON or reset
     */
    float LWM2M_RESOURCE( 5601, Min_Measured_Value );
    /**
     *  @brief Min Range Value
     *
     *  Operations: read
     *
     *  Not mandatory
     *
     *  Instance type: single
     *
     *  Units: defined by 'units' resource.
     *
     *  Type: float
     *
     *  The minimum value that can be measured by the sensor
     */
    float LWM2M_RESOURCE( 5603, Min_Range_Value );
    /**
     *  @brief Max Measured Value
     *
     *  Operations: read
     *
     *  Not mandatory
     *
     *  Instance type: single
     *
     *  Units: defined by 'units' resource.
     *
     *  Type: float
     *
     *  The maximum value measured by the sensor since power ON or reset
     */
    float LWM2M_RESOURCE( 5602, Max_Measured_Value );
} LWM2M_IPSO_Temperature_T;

/**  IPSO Temperature */
#define LWM2M_IPSO_Temperature LWM2M_OBJECT_ID( 3303 )
```

C Structure Used for the LWM2M Temperature Object in ScriptML

If we examine the code in a little more detail, we can make a few observations.

1. The Temperature Object is defined as a C structure
2. The structure contains a set of resources which are defined as a range of C types such as strings, floats (other objects will also include boolean and Int types)
3. In this example the resources are single instance, not mandatory (except for the sensor value)
4. The resources are defined using a C macro `LWM2M_RESOURCE` which is used to link the resource to the client domain via a ScriptML intrinsic (see Ref [2] for an introduction to ScriptML intrinsics). Within this resource definition the Resource ID is defined as well as the Resource name that will be used within the script.
5. Finally the Object ID is defined and will be used internally within the ScriptML core to link the Object in the ScriptML domain to the code in the client domain.

We won't explore the implementation of a sensor using this object here, instead we will look at some other types of object first and then look at a few real examples.

OIOL Client Push Text Object – ID 7019

The next example we will consider is the Client Push Text Object. This object was created by OpenIO Labs to allow the user to push text to the server. The object identifiers are in a number range that is reserved for vendor specific implementations, so that they will not clash with IPSO defined objects and resources.

The code snippet from the lwm2m.h header file is presented below.

```
/**
 *  @brief OpenIO Client-pushed text object
 *
 *  Mandatory
 *
 *  Instance type: single
 *
 *  Allows the client to push text objects to the server, and optionally,
 *  to the UI
 */
typedef struct
{
    /**
     *  @brief ObjectID
     *
     *  Operations: write
     *
     *  Not mandatory
     *
     *  Type: string
     *
     *  Instance type: single
     *
     *  This is the text object ID which is written back immediately after the
     *  client pushes a text object. This allows the client to overwrite a
     *  text object if necessary.
     */
    char * LWM2M_RESOURCE( 1, ObjectID );
    /**
     *  @brief Object
     *
     *  Operations: read, write
     *
     *  Mandatory
     *
     *  Type: string
     *
     *  Instance type: single
     *
     *  A JSON-blob, with the following parameters: {objectName: "A name for
     *  the object", objectID: "Only used when OVERWRITING an existing object,
     *  see resource 1.", objectText: "The content", tags:
     *  ["a","list","of","tags"], pushToUI: true}
     */
    char * LWM2M_RESOURCE( 0, Object );
} LWM2M_OpenIO_Clientpushed_text_object_T;

/**  OpenIO Client-pushed text object */
#define LWM2M_OpenIO_Clientpushed_text_object LWM2M_OBJECT_ID( 7019 )
```

C Structure Used for the OIOL Client Push Text Object in ScriptML

The structure of the Client Push Text Object is very similar to the Temperature object we saw previously. In this case it uses two string resources, and each string resource is used as a container for a JSON object that defines the details of the Text object.

In a following section we will see an example of the use of this Object.

OIOL Motor Control Object – ID 7017

The next example we will consider is the Motor Control Object. This object was created by OpenIO Labs to allow the user to control a motor from the server.

The code snippet for the Motor Control object is presented below.

```
/**
 *  @brief OpenIO Motor Control
 *
 *  Not mandatory
 *
 *  Instance type: multiple
 *
 *  Actuator object to control a motor
 */
typedef struct
{
    /**
     *  @brief State
     *
     *  Operations: read, write
     *
     *  Mandatory
     *
     *  Type: boolean
     *
     *  Instance type: single
     *
     *  Motor state (on/off)
     */
    int LWM2M_RESOURCE( 1, State );
    /**
     *  @brief Velocity
     *
     *  Operations: read, write
     *
     *  Mandatory
     *
     *  Type: integer
     *
     *  Instance type: single
     *
     *  Motor velocity
     */
    int LWM2M_RESOURCE( 0, Velocity );
    /**
     *  @brief Direction
     *
     *  Operations: read, write
     *
     *  Mandatory
     *
     *  Type: boolean
     *
     *  Instance type: single
     *
     *  Direction (true == clockwise)
     */
    int LWM2M_RESOURCE( 2, Direction );
} LWM2M_OpenIO_Motor_Control_T;
```

```
/**  OpenIO Motor Control */
#define LWM2M_OpenIO_Motor_Control LWM2M_OBJECT_ID( 7017 )
```

C Structure Used for the OIOL Motor Control Object in ScriptML

Again, the structure for the Motor Control object is very similar to the ones we have seen previously. In this case a number of Integer types are defined. Two are used as booleans and the third is used to set the Motor Control velocity.

We will see an example of the use of this object in a later section.

LWM2M Specific Methods in C

In addition to the LWM2M Objects and Resources other methods are required to allow ScriptML access to the server via the client. At present only a single method is defined, namely the Register method, however, it is expected that future revisions will also introduce additional methods as the need arises.

LWM2M Register Method

The code snippet presented below is a C macro that links the ScriptML C register method with the ScriptML core via the use of ScriptML intrinsics.

```
/** Register an LWM2M object with the server
 * Note: ID is deduced as the identifier index/OMI handle for OBJ
 */
#define sml_lwm2m_register_object( OBJ ) \
{ \
  ___OMISend(request:"SM2M_AddObjects", client_handle:-1,
snippet_payload:___ISeq(OBJ), int_payload:___ISeq()); \
  ___ListType_T(element_type:___NULL(), size:___NULL()) l = ___Receive(); \
  assert( l[0]=="OK" ); \
}
```

ScriptML LWM2M Register Method

The details presented above are purely for information as the end user will have no visibility of ScriptML intrinsics or the need to use them.

From a users perspective, the method would be invoked in a manner such as this (using the Motor Control object as an example).

```
/* Declare an LWM2M object as a ScriptML variable */
LWM2M_OpenIO_Motor_Control_T LWM2M_OpenIO_Motor_Control;

/* Register our object with server, the LWM2M_OpenIO_Motor_Control
 * is now externally readable and modifiable */
```

```
sml_lwm2m_register_object(LWM2M_OpenIO_Motor_Control);
```

Example Use of the ScriptML LWM2M Register Method using C

From the example above, we can see how simple it is from the end-users perspective to create and register, what is quite a complex LWM2M object with the server. If this script were run on the client, a LWM2M Motor Control object would appear on the server.

In a later section we will explore an example that uses this object.

Chapter 4: LWM2M Support Using Python

In this section we will examine the LWM2M support for ScriptML with the source file written in Python. We will not repeat all of the discussions on the objects types and details that we presented in the previous Chapter, instead we will focus on the code segments used for a Python implementation.

In the Python case, we define a Python package (lwm2m.py). Like the C case this Python package is auto-generated in the OpenIO Labs build system. The Objects and the Resources that are defined in the package are identical to the C header file, the difference is that the package represents the Python syntax that is required.

Accessing LWM2M Objects and Resources in Python

As in the C case, we will examine three different Objects and their resources starting with the Temperature object.

IPSO Temperature Object – ID 3303

The code snippet from the Python package that supports the LWM2M Temperature object is presented below.

```
IPSO_Temperature_Sensor_Value = 5700
IPSO_Temperature_Sensor_Units = 5701
IPSO_Temperature_Reset_Min_and_Max_Measured_Values = 5605
IPSO_Temperature_Max_Range_Value = 5604
IPSO_Temperature_Min_Measured_Value = 5601
IPSO_Temperature_Min_Range_Value = 5603
IPSO_Temperature_Max_Measured_Value = 5602
##
#  @brief IPSO Temperature
#
#  @return new LWM2M object
#
#  Not mandatory
#
#  Instance type: multiple
#
#  Description: This IPSO object should be used with a temperature sensor
#  to report a temperature measurement.  It also provides resources for
#  minimum/maximum measured values and the minimum/maximum range that can
#  be measured by the temperature sensor. An example measurement unit is
#  degrees Celsius (ucum:Cel).
#
def create_IPSO_Temperature():
    return {
        IPSO_Temperature_Sensor_Value: float(), # Sensor Value
        IPSO_Temperature_Sensor_Units: str(), # Sensor Units
        IPSO_Temperature_Reset_Min_and_Max_Measured_Values: str(), # Reset
        IPSO_Temperature_Max_Range_Value: float(), # Max Range Value
        IPSO_Temperature_Min_Measured_Value: float(), # Min Measured Value
        IPSO_Temperature_Min_Range_Value: float(), # Min Range Value
        IPSO_Temperature_Max_Measured_Value: float(), # Max Measured Value
    }

##  IPSO Temperature
id_IPSO_Temperature = 3303
```

Python Method used for the LWM2M Temperature Object in ScriptML

We won't explore the details of the method presented above, other than to say it is functionally equivalent to the C example. It uses a Python method to create a temperature object that includes a number of resource elements with appropriate identifiers assigned to all instances.

OIOL Client Push Text Object – ID 7019

The next example is the Client Push Text object. The code snippet from the Python LWM2M package is shown below.

```
OpenIO_Clientpushed_text_object_ObjectID = 1
OpenIO_Clientpushed_text_object_Object = 0
##
#  @brief OpenIO Client-pushed text object
#
#  @return new LWM2M object
#
#  Mandatory
#
#  Instance type: single
#
#  Allows the client to push text objects to the server, and optionally,
#  to the UI
#
def create_OpenIO_Clientpushed_text_object():
    return {
        OpenIO_Clientpushed_text_object_ObjectID: str(), # ObjectID
        OpenIO_Clientpushed_text_object_Object: str(), # Object; RW
    }

##  OpenIO Client-pushed text object
id_OpenIO_Clientpushed_text_object = 7019
```

Python Method used for the OIOL Client Push Text Object in ScriptML

OIOL Motor Control Object – ID 7017

Finally we have the OIOL Motor Control object defined in the LWM2M Python package.

```
OpenIO_Motor_Control_State = 1
OpenIO_Motor_Control_Velocity = 0
OpenIO_Motor_Control_Direction = 2
##
#  @brief OpenIO Motor Control
#
#  @return new LWM2M object
#
#  Not mandatory
#
#  Instance type: multiple
#
#  Actuator object to control a motor
#
def create_OpenIO_Motor_Control():
    return {
        OpenIO_Motor_Control_State: bool(), ##< State; RW; boolean
        OpenIO_Motor_Control_Velocity: int(), ##< Velocity; RW; integer
        OpenIO_Motor_Control_Direction: bool(), ##< Direction; RW; boolean
    }

##  OpenIO Motor Control
id_OpenIO_Motor_Control = 7017
```

Python Method used for the OIOL Motor Control Object in ScriptML

LWM2M Specific Methods in Python

Similar to the C case, we also need a Python method that is used to register the Objects with the server.

LWM2M Register Method

In this example, we present a simple code snippet that illustrates how to create the LWM2M Motor Control object in Python, and then go on to register it with the server.

```
# Create the Motor Control object
my_motor = lwm2m.create_OpenIO_Motor_Control()

# Register the motor control object with the server
lwm2m.register_object_id(lambda: my_motor, lwm2m.id_OpenIO_Motor_Control)
```

Example Use of the ScriptML LWM2M Register Method using Python

Chapter 5: Example Application in C

In this chapter we will create a simple application that uses some of the LWM2M Objects and Resources that we have seen.

The example will be a Motor Control object that receives input from the user via their browser to the server, a ScriptML script written in C that controls the motor and sends data to the server on the motor velocity.

The idea is that the user will set whether the motor is active or not and the direction of the rotation. The script will then either increase or decrease the motor velocity accordingly. The motor velocity will be sent to the server to display to the user.

What we will obtain is the user interacting with the script on the device via the UI on the browser, the script will respond to these changes and perform some calculations and update the server with these new values. The user will then see the affects of these changes on the server.

Although a simple example, it demonstrates the power of ScriptML it's interactions with the server, the users interactions with the server via the browser and how all of these elements fit together in a seamless manner.

The code for the example is presented below with in-line comments to explain the purpose for the different code segments.

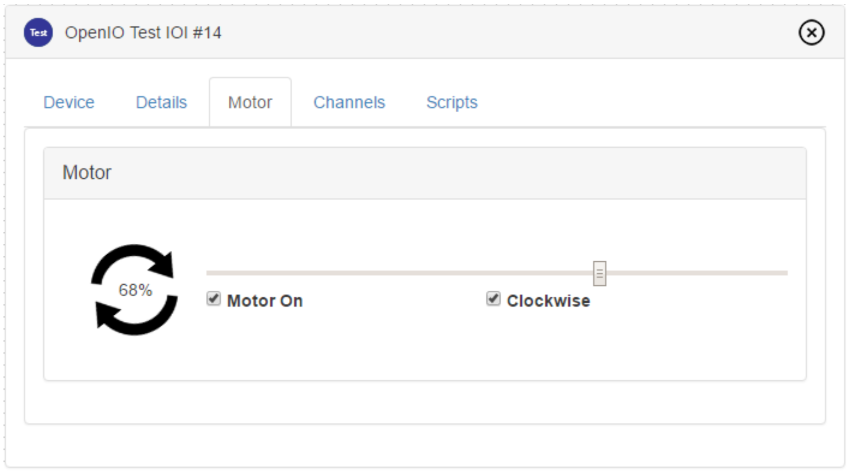
Within the code we can see how the LWM2M object is created, registered and then used by the script.

```
#include <lwm2m.h>
#include <test_print.h>

/* Declare an LWM2M object as a ScriptML variable */
LWM2M_OpenIO_Motor_Control_T LWM2M_OpenIO_Motor_Control;

int main()
{
    /* Register our object with the server */
    sml_lwm2m_register_object(LWM2M_OpenIO_Motor_Control);

    while(1)
    {
        /* Loop to slow down the rate we push to the server */
        for( int i=0; i<1000; i++ ){}

        /* Keep on printing the "state" resource of the object */
        printint( LWM2M_OpenIO_Motor_Control.State );
        printint( LWM2M_OpenIO_Motor_Control.Direction );
        printint( LWM2M_OpenIO_Motor_Control.Velocity );

        /* Test that the Motor Control object is active */
        if( LWM2M_OpenIO_Motor_Control.State )
        {
            /* Move the "velocity" slider */
            int v = LWM2M_OpenIO_Motor_Control.Velocity;

            /* Check the motor direction and set speed accordingly */
            if( LWM2M_OpenIO_Motor_Control.Direction )
            {
                v++;
            }
            else
            {
                v--;
            }

            /* If we have reached Max or Min set to Max or Min */
            if( v < 0 )
            {
                v = 0;
            }
            else if( v > 100 )
            {
                v = 100;
            }
            /* Push the new velocity value to the server */
            LWM2M_OpenIO_Motor_Control.Velocity = v;
        }
    }
    return 0;
}
```

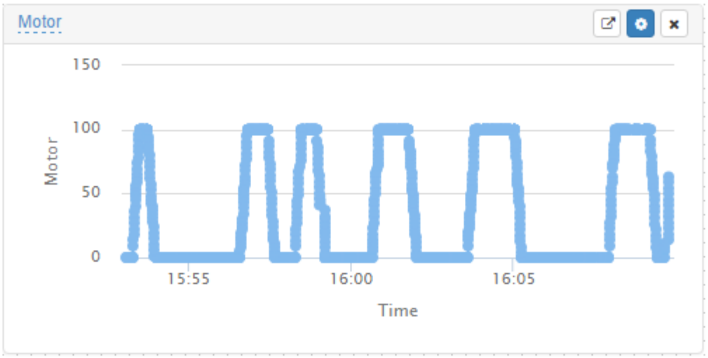
Simple Motor Control Example Executed in ScriptML from a C source File

Assuming that we have created this C source file we can upload this to the OpenIO Labs server and deploy it on our target device. For details on doing this, refer to the UG102 User Guide Ref [5].

If we now look at what we see on the users browser when the script is running.



Here we see the view of the motor control from the users perspective. The user can switch on and off the motor and also change from clockwise to anti-clockwise.



Here we see the graph of the motor speed with time. You can see the points where the user has changed the direction of rotation as it moves between 100 and 0.

Chapter 6: Example Application in Python

In the code segment below, we can see the equivalent Python script with in-line comments to explain the different sections.

The output on the users browser will be identical to the C script we saw earlier.

```
import lwm2m

# Create the motor control object
my_motor = lwm2m.create_OpenIO_Motor_Control()
# Register the object with server
lwm2m.register_object_id(lambda: my_motor, lwm2m.id_OpenIO_Motor_Control)

while(1):

    # Loop to delay server upload rate
    for i in range(1000):
            pass
    # Keep on printing the "state" resource of the object (id=1)
    print my_motor[lwm2m.OpenIO_Motor_Control_State]
    print my_motor[lwm2m.OpenIO_Motor_Control_Direction]
    print my_motor[lwm2m.OpenIO_Motor_Control_Velocity]

    # Check the motor is still active
    if my_motor[lwm2m.OpenIO_Motor_Control_State]:

        # Move the "velocity" slider
        v = my_motor[lwm2m.OpenIO_Motor_Control_Velocity]

        # Based on the direction the motor is set to rotate
        # increase or decrease the velocity
        if my_motor[lwm2m.OpenIO_Motor_Control_Direction]:
            v+=1
        else:
            v-=1

        # heck if we have reached min or max and if so set to min or max
        if v < 0:
            v = 0
        elif v > 100:
            v = 100

        # Push the new velocity value to the server
        my_motor[lwm2m.OpenIO_Motor_Control_Velocity] = v
```

Simple Motor Control Example Executed in ScriptML from a Python source File

Additional Resources

References

The following documents and references are cited within this guide:

1. UG101 – OpenIO Labs System Architecture
2. UG401 – ScriptML System Overview
3. LWM2M Technical Specification – OMA-TS-LightweightM2M-V1_0-20141126-C
4. IPSO SmartObject Guideline - Smart Objects Starter Pack1.0
5. UG102 – Accessing the OpenIO Labs System