

OpenIO Labs - ScriptML

ScriptML Examples in C and Python

UG402 (v2017.1) April 3, 2017

Revision History

Date	Version	Revisions
3/4/2017	2017.1	<i>Initial release.</i>

Table of Contents

Revision History	2
Chapter 1: Introduction	4
Chapter 2: Hello ScriptML	5
Hello ScriptML in C.....	5
Hello ScriptML in Python.....	10
Chapter 3: Adding Hardware to the IOI	13
DAC Driver Script in C.....	14
DAC Driver Script in Python.....	15
ADC Driver Script in C.....	17
ADC Driver Script in Python.....	20
Chapter 4: Connecting to the Server	22
<i>ADC and DAC Driver and Connecting to the Server – in C.....</i>	<i>22</i>
<i>ADC & DAC Driver Connecting to the Server – in Python.....</i>	<i>28</i>
Summary.....	29
Conclusions	30
Appendix	31
C – Header and Python Package Files.....	31
Additional Resources	39
References.....	39

Chapter 1: Introduction

This User Guide outlines the steps to follow to create some simple ScriptML scripts and download them to the edge device (IOI). The Scripts are written in C and Python, although similar scripts could also be written in C++, Java and Golang.

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 [1,2,3] 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 from the web link provided by Ref [5].

To start, as in all first steps when learning new programming languages, we will start with a simple text application. Following on from that we will start to interact with real hardware devices attached to the IOI. In these examples, we will focus on the use of an Analogue to Digital Converter (ADC) and a Digital to Analogue Converter (DAC) accessed via the edge point device using the I2C protocol. For those unfamiliar with I2C, please refer to the weblink presented in Ref [4].

The User Guide is written in a self-paced manner, with each section building on the previous sections. If you are not familiar with the OpenIO Labs system, it is recommended that you work through each of these steps before moving to the next.

The hardware examples (ADC and DAC) utilise header files (C) and Packages (Python) that are tailored to these specific devices. In the Appendix you will find an introduction to these files used in the examples. A separate User Guide (UG403 – Ref [7]) goes through the steps to create a header file / package for a new device. The steps are simple and in many cases existing device drivers (e.g. for the Linux OS) can be taken and adapted for use within the OpenIO Labs ScriptML system. In doing so, you are able to take advantage of the many diverse devices in existence and bring them into your system design and write scripts in C or Python that are transferred to your edge device using ScriptML.

A number of the examples also make explicit use of the LWM2M Objects and protocols to allow the edge device to inter-connect to the OpenIO Labs server. The OpenIO Labs server is compatible with the LWM2M protocol and the LWM2M objects defined and supported by the standard. Use of some of these LWM2M Objects and the OpenIO Labs extensions is described in this User Guide. For a more detailed description of the LWM2M objects that ScriptML supports in C and Python, the interested reader is referred to Ref [8] in which a more explicit description of the different types of devices is presented.

Chapter 2: Hello ScriptML

We will start this chapter with some simple examples in C and Python that will allow you to write scripts, upload them to the server, and then execute them on the server. As with most users new to a computer language, we will start with a simple “Hello World” text example.

Hello ScriptML in C

We will start with the C version of this simple script. Using a text editor, or IDE, create the following script on your local host machine. Name the script something like myFirstScript.c.

```
/* Test Script that Pushes a JSON text Blob using C */
#include <lwm2m.h> // lwm2m.h is auto-generated
#include <stdint.h>
#include <test_print.h>
#include <stdio.h>

/* Define an instance of a client Push LWM2M Text Object */
LWM2M_OpenIO_Clientpushed_text_object_T LWM2M_OpenIO_Clientpushed_text_object;

int main()
{
    /* Define an ObjectID - required to be defined, but not used in this example */
    char *objId = "{\\""}";

    /* Link the text fields to the Client text Object to the Server */
    LWM2M_OpenIO_Clientpushed_text_object.ObjectID = objId;
    LWM2M_OpenIO_Clientpushed_text_object.Object = "";

    /* Register the Client Push Text Object with the Server */
    sml_lwm2m_register_object(LWM2M_OpenIO_Clientpushed_text_object);

    /* Define the text in json format */
    char *strToSend = "{\\"objectName\\": \\"ObjectText\\", \\"objectText\\": \\"Hello ScriptML\\", \\"tags\\": [\\"my_text_tag\\"], \\"pushToUI\\": true}";

    /* Write to the registered LWM2M resource, to push to the text object to the server */
    LWM2M_OpenIO_Clientpushed_text_object.Object = strToSend;

    return 0;
}
```

Exploring the Script

Let us examine the details of this C script a line at a time (excluding the comments).

```
#include <lwm2m.h> // lwm2m.h is auto-generated
```

This first line pulls in all of the LWM2M objects. A description of the LWM2M objects can be found in the reference manuals that can be found from Ref [5] and the OpenIO Labs support for these in Ref [8]. In addition to the standard LWM2M objects, there are some OpenIO Labs specific objects

that have been defined, and are used by the system.

Next we need to define an object that is referred to as the Client Pushed Text Object.

```
LWM2M_OpenIO_Clientpushed_text_object_T LWM2M_OpenIO_Clientpushed_text_object;
```

The Client Pushed Text Object is defined using a C struct in the LWM2M header file. So first we create an instantiation of that C struct. The name we use is similar to the struct name, but you could use a shorter name if preferred.

Next we come to the main loop, and the first part of the main loop.

```
char *strToSend = "{\"objectName\": \"ObjectText\", \"objectText\": \"Hello  
ScriptML\", \"tags\": [\"my_text_tag\"], \"pushToUI\": true}";  
char *objId = "{}";
```

Here we are defining two C pointers to json text objects. The format looks a little strange because we need to include quotation marks in the object. To do this in C, we need to “escape” the quotation marks, such that “ becomes \”. The format of the string when it is converted to a C string 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

The second pointer is to an ObjectID string that is used to identify the text push object should we desire to add to the object in a subsequent iteration, but here, we simply set that to a null string.

Next we will link the two text object pointers to the LWM2M Client Push Text object that we will send to the server.

```
LWM2M_OpenIO_Clientpushed_text_object.Object = strToSend;  
LWM2M_OpenIO_Clientpushed_text_object.ObjectID = objId;
```

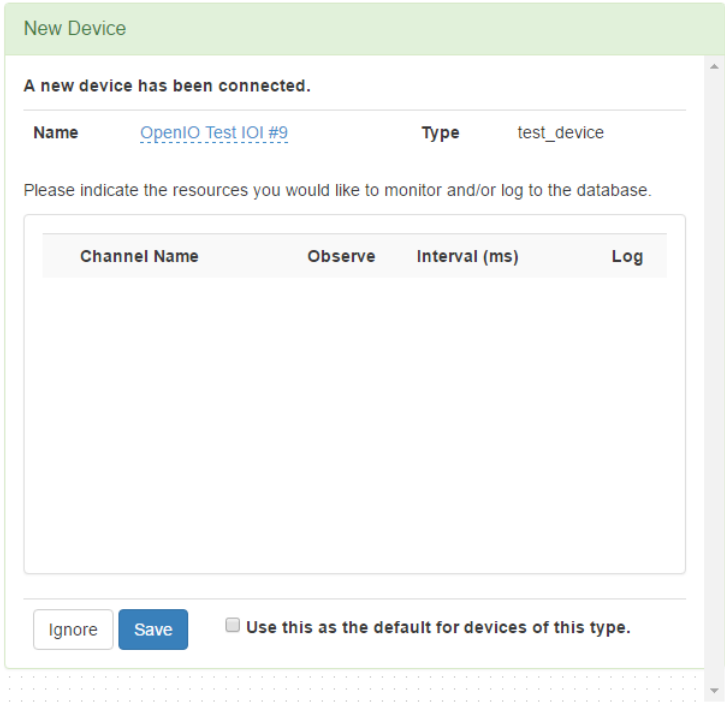
Finally, we register the Client Pushed Text Object with the server, this is a ScriptML internal function that will register the Client Pushed Text Object with the server using the LWM2M protocol:

```
sml_lwm2m_register_object(LWM2M_OpenIO_Clientpushed_text_object);
```

Running the Script

Now that we have completed our first ScriptML script in C, we can upload it to the server. To do this we will need first to log into the server, and then power on the OpenIO Labs edge device (IOI). For details of logging into the server and accessing the server via a browser, see Ref [2].

When we have switched on the IOI and have logged into the server we will see a screen something like this on the browser that is accessing the server:

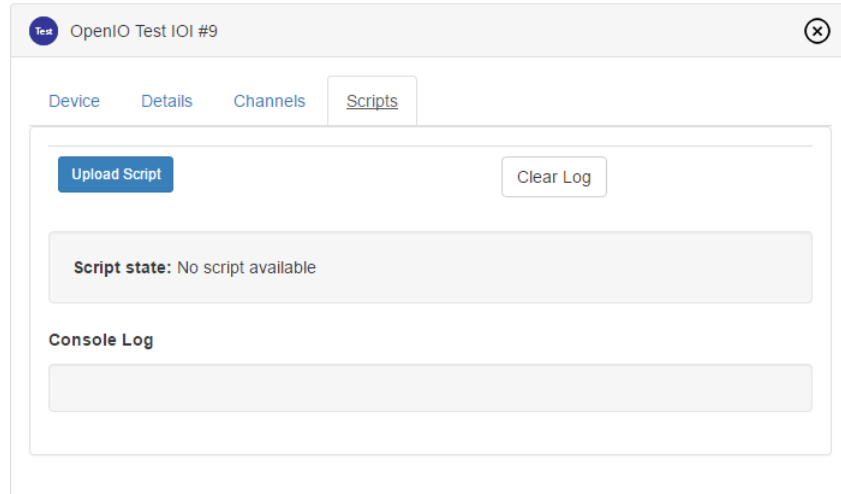


The screenshot shows a web form titled "New Device" with a green header. Below the header, a message states "A new device has been connected." The form contains two fields: "Name" with the value "OpenIO Test IOI #9" and "Type" with the value "test_device". Below these fields, a prompt asks the user to "Please indicate the resources you would like to monitor and/or log to the database." This is followed by a table with four columns: "Channel Name", "Observe", "Interval (ms)", and "Log". The table is currently empty. At the bottom of the form, there are two buttons: "Ignore" and "Save", and a checkbox labeled "Use this as the default for devices of this type." which is currently unchecked.

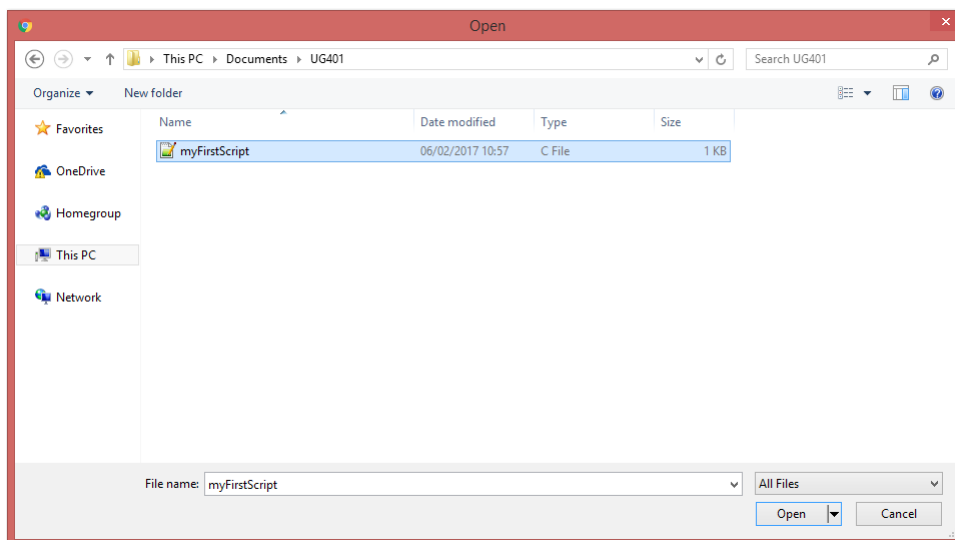
This is the IOI registering with the server. You should only see this the first time that you register the device, unless you ask the server to "Forget" the device.

In this instance, click on "Save" so that the server will remember the IOI for future sessions.

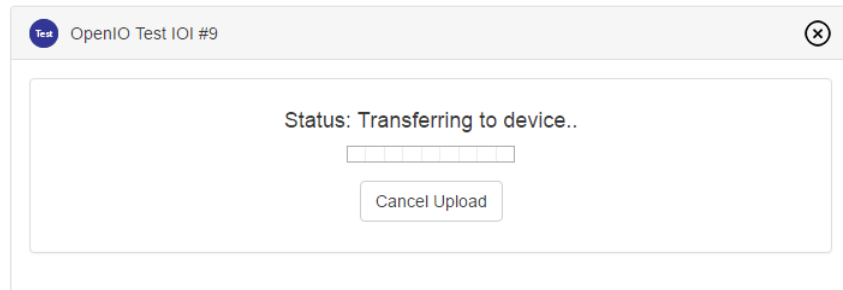
If you now open the device by clicking on the device from the left hand side of the browser and select the scripts tab, you should see something like this:



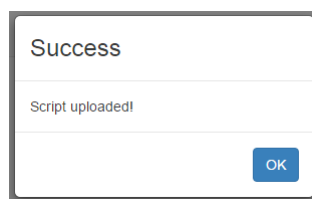
From this screen, you can upload the script to the server. To do this click on the Upload Script button. You will be asked to browse to the directory where you have saved your C script. Select the C script and click Open.



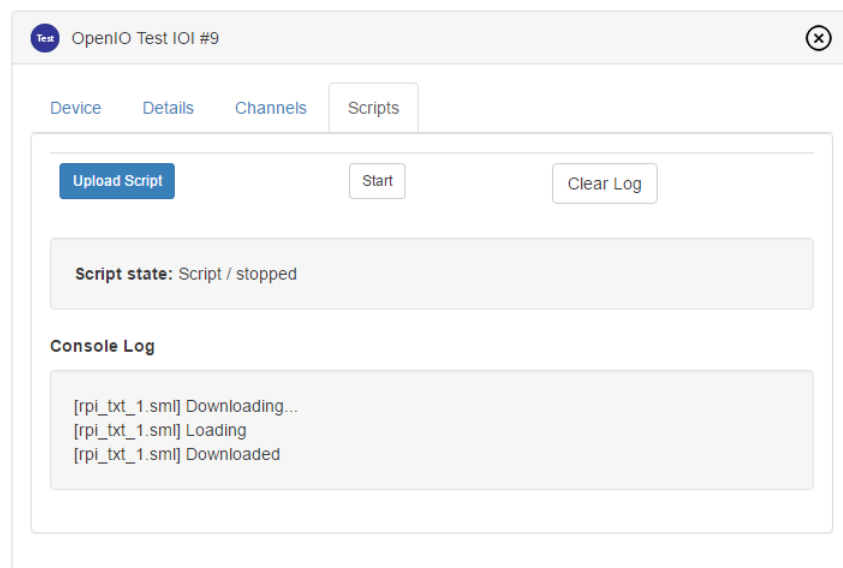
Depending on the network speed, you may see the following screen as the script is uploaded to the server.



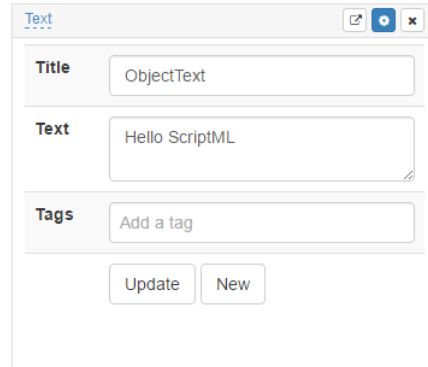
Under the hood, the server is uploading the C script and translating it into a ScriptML file that will be understood by the IOI device. The server will then download the ScriptML file to the device ready to start its execution. When it is complete, you will see the following pop-up window in your browser.



Click on OK and you will see this screen:



When you are ready, you can start the script, by clicking on the Start button. After doing this, the script will execute and you will see the following pop-up window in the browser window:



Here we can see the text object that we have created and the text that we have added to the text object.

Summary

So, to summarise, we have done the following:

1. Create a C script in an editor
2. Powered on the IOI device which registered with the server using LWM2M
3. Upload the script to the server and then down to the device
4. Execute the script

Hello ScriptML in Python

Now we can look at the Python version of this simple script. Using a text editor, or IDE, create the following script on your local host machine. Name the script something like myFirstScript.py.

```
# Test Script that Pushes a JSON text Blob using Python
import lwm2m

# Create the Client Push Text Object
my_text_object = lwm2m.create_OpenIO_Clientpushed_text_object()

# Register our object with server
lwm2m.register_object_id(lambda: my_text_object, lwm2m.id_OpenIO_Clientpushed_text_object)

# Define the json object with the text
text_to_send = "{\"objectName\": \"ObjectText\", \"objectText\": \"Hello ScriptML from Python\", \"tags\": [\"my_text_tag\"], \"pushToUI\": true}"

# Push the text to the server
my_text_object[lwm2m.OpenIO_Clientpushed_text_object_Object] = text_to_send
```

Exploring the Script

Let us examine the details of this Python script a line at a time (excluding the comments).

```
import lwm2m
```

The Python import pulls in the LWM2M package that is auto-generated in the OpenIO Labs system from the json LWM2M objects that the system supports. With the package imported we can now start to use the objects from within Python (see Ref [8] for more details on LWM2M objects supported via ScriptML).

Next we need to create the Client Push Text Object, and this is done as follows:

```
my_text_object = lwm2m.create_OpenIO_Clientpushed_text_object()
```

This will instantiate the text object that we will use to push the text to the server.

Now, we will register the text object with the server as follows:

```
lwm2m.register_object_id(lambda: my_text_object, lwm2m.id_OpenIO_Clientpushed_text_object)
```

Note that we are registering the object using the identity of the object. The object identity is defined in the LWM2M python package that we imported previously.

We can now create the json text object that we want to push to the server:

```
text_to_send = "{\"objectName\": \"ObjectText\", \"objectText\": \"Hello ScriptML from Python\", \"tags\": [\"my_text_tag\"], \"pushToUI\": true}"
```

You will notice that the format and meaning of the json text object is the same as the one that was used in the C script earlier. The use and meaning of the different identifiers is the same here as it is there.

Finally, we can link the json text object to the client push text object and push the text to the server, this is done as follows:

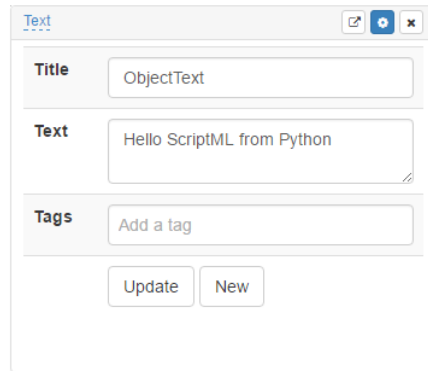
```
my_text_object[lwm2m.OpenIO_Clientpushed_text_object_Object] = text_to_send
```

Running the Script

Now that we have completed our Python version of the text push object, we can run the script on the IOI device.

To do this, we follow exactly the same procedure as in the C script example, except now we will upload the python script in place of the C script. The OpenIO Labs server will identify that it is a Python script from the .py file extension and will translate the Python application into native ScriptML code.

When we load and start the execution of this script we will see the following on the server.



The screenshot shows a web form titled "Text" with a light gray border and standard window controls (minimize, maximize, close) in the top right corner. The form contains three input fields: "Title" with the value "ObjectText", "Text" with the value "Hello ScriptML from Python", and "Tags" with the placeholder text "Add a tag". Below these fields are two buttons: "Update" and "New".

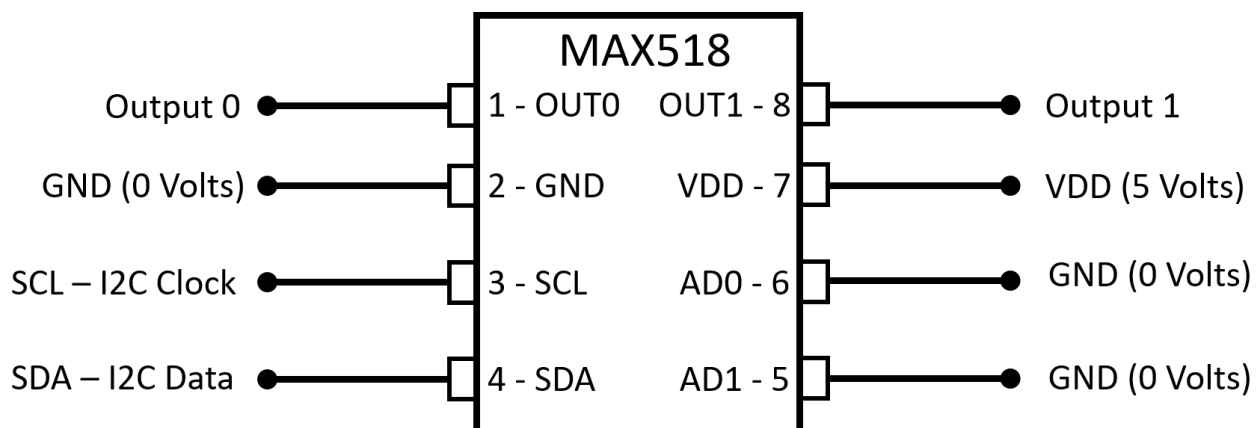
This is the Python script converted in ScriptML and executing on the IOI device.

Chapter 3: Adding Hardware to the IOI

Now that we have our first sample scripts running, we can move a little deeper and start to add some real hardware to the system. We will start with a simple device, a Digital-to-Analogue converter (DAC).

For those not familiar with a DAC, it is a device that accepts a digital number (represented as binary) and converts it to an analogue signal, in this case it is a voltage. In this example we will be using a Maxim MAX518 DAC (see Ref [6] for details and datasheets).

The MAX518 is an 8 bit DAC which we will operate from a 5 Volts power supply. This means that a binary value of 0 is 0 Volts and a value of 255 is 5 Volts. The MAX518 is connected to the IOI using an I2C interface. For details of the I2C interface for the IOI that you are using refer to the User Guide for that IOI device. As with all I2C devices, you need to remember to include a pull-up resistor from the clock and data lines to the 5 Volts power rail. Typically a 2.2k Ohm resistor will be sufficient.



So we will assume that you have a MAX518 connected to the IOI with the 0 Volts , 5 Volts and SDA and SCK lines connected also (as illustrated in the figure above). We can start to look at the scripts that we will use to drive the DAC. As before we will start with the C version of the script, before moving on to the Python version.

In this first version of the DAC driver scripts in C and Python, we will focus on driving the DAC, but not worry about connecting the DAC to the server to view the results. In a later example we will include both the server connection via the LWM2M protocol as well as the control of the DAC and an ADC.

DAC Driver Script in C

We will start with the C version of the DAC driver script. Using a text editor, or IDE, create the following script on your local host machine. Name the script something like myFirstDAC.c.

```
#include <scriptml/i2c.h>
#include <scriptml/max518.h>

int main()
{
    /* Create an instance of the MAX518 device,
    and link it to the I2C interface */
    sml_max518_t dac = sml_max518_create( sml_i2c_open( "/dev/i2c-1" ), 0 );

    /* Next a simple loop that will create a ramp waveform for
    the two outputs.
    Output 0 will increase from 0, and output 1 will decrease from 255.
    At the output of the DAC we should see two triangular waveforms
    in opposite directions */
    int iLoop=0;
    while(1)
    {
        /* The ramp up cycle for Out 0, reversed for Out 1 */
        for( ; iLoop<255; iLoop++ )
        {
            /* Push the digital level to the DAC for Out 0 and Out 1 */
            sml_max518_set_level( dac, 0, iLoop );
            sml_max518_set_level( dac, 1, 255- iLoop );
        }

        /* The ramp down cycle for Out 0, reversed for Out 1 */
        for( ; iLoop>0; iLoop-- )
        {
            sml_max518_set_level( dac, 0, iLoop );
            sml_max518_set_level( dac, 1, 255- iLoop );
        }
    }
    return 0;
}
```

Example DAC script written in C

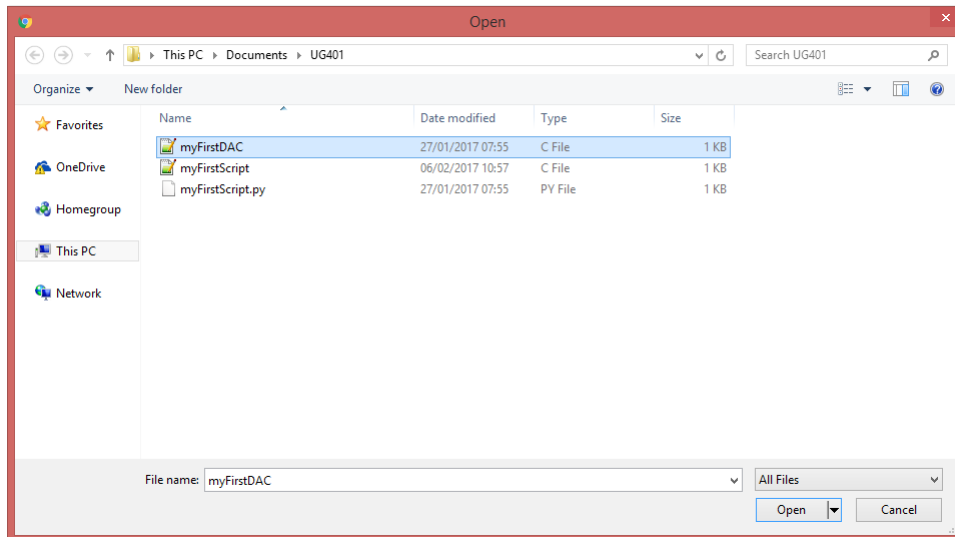
The example uses the MAX518.h header file that is presented in the Appendix. The header file defines the methods that create an instance of the device and also allow the level to be set on one of the two channels for the device.

The ScriptML example starts by creating an instance of the DAC device. Which defines the I2C interface that will be used.

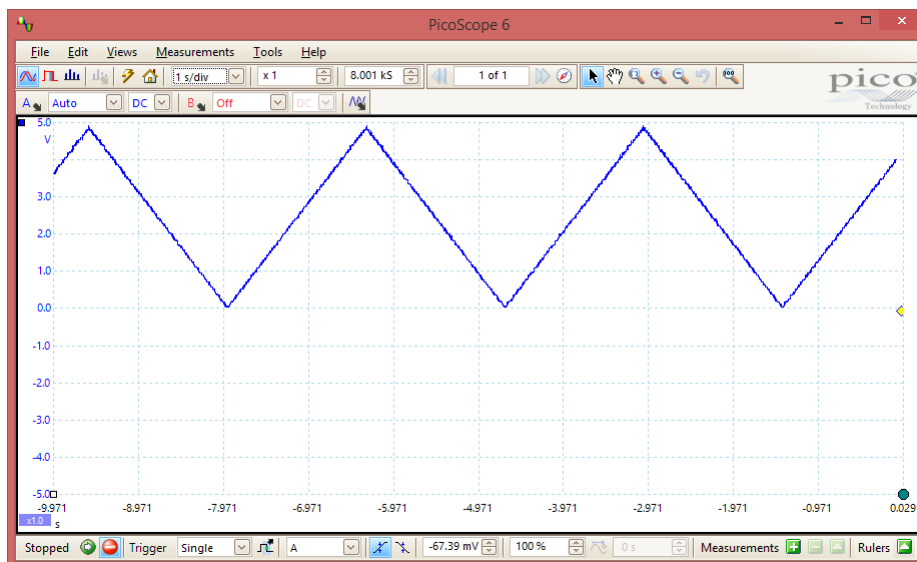
The next two sections are two loops. The first is an incrementing loop (a ramp up for Output 0 and a ramp down for Output 1). The second loop is a decrementing loop (a ramp down for Output 0 and a ramp up for Output 1). We combine these together in an infinite loop and expect to see a triangular waveform at the outputs, with Output 1 and inverted version of Output 0.

To load and run the script, follow the same procedure as defined for the Client Push Text script,

except we are loading the DAC C script that we have just written as shown below after pressing script Upload on the browser.



The image that we see on an Oscilloscope is shown in the picture below for Output 0.



Output 0 from the MAX518 viewed on an Oscilloscope driven by ScriptML using C source

DAC Driver Script in Python

The DAC driver in Python is illustrated below. The driver also uses the MAX518 device. The Python

package for this device is presented in the Appendix.

```
import max518
import i2c

# Create an instance of the DAC device
dac = max518.device( i2c.device("/dev/i2c-1"), 0 )

# Two loops, first increment Out 0, decrement Out 1
# Second decrementing Out 0 and incrementing Out 1
while True:
    for iLoop in range(256):
        dac.set_level( 0, iLoop )
        dac.set_level( 1, 255- iLoop )
    for iLoop in range(255, 0, -1):
        dac.set_level( 0, iLoop )
        dac.set_level( 1, 255- iLoop )

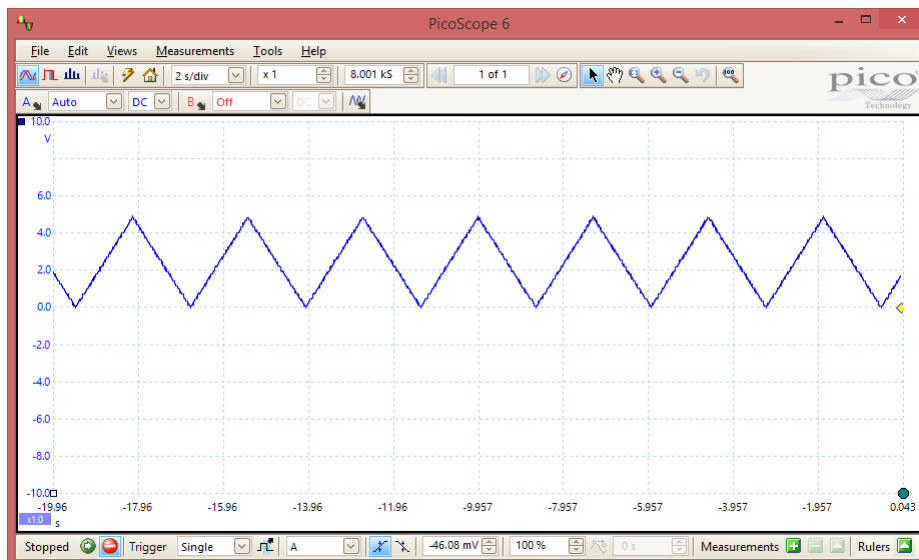
del dac
```

Example DAC script written in Python

The driver starts by creating an instance of the DAC, and then using two loops increments the Output 0 while decrementing Output 1. In the second loop this is reversed, and we produce a triangular waveform.

The Python Script is loaded from the server to the device, and from the server interface the script is started and the triangular wave can be viewed.

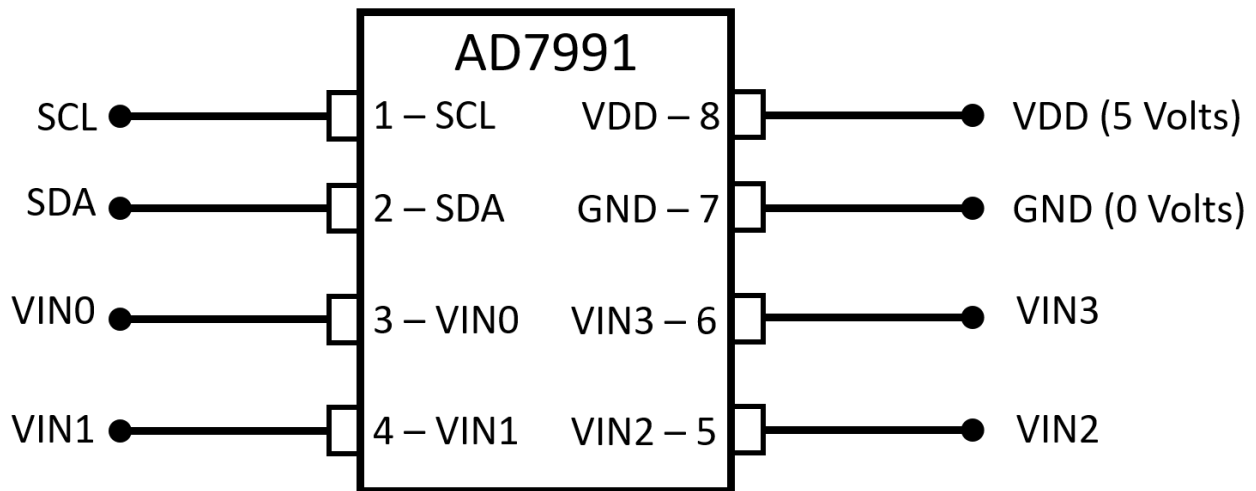
The output of Output 0 was viewed on an oscilloscope and illustrated in the picture below.



Output 0 from the MAX518 viewed on an Oscilloscope driven by ScriptML using Python source

ADC Driver Script in C

Now, we want to add an additional device to the IOI also using the I2C interface. In this case, the device that we will use is an Analog Devices AD7991 Analogue to Digital Converter (ADC). The AD7991 is a 12 bit ADC and in this instance is connected to a 5 Volts power supply. The recommended connections for this device are illustrated in the figure below.



When an analogue voltage is applied to the ADC input it is digitised into a digital version. A digital value of 0 corresponds to 0 Volts, and a digital value of 4095 corresponds to 5 Volts. For more details of the AD7991 ADC refer to Ref [9].

Next we will create a C version of an ADC driver script. Using a text editor, or IDE, create the following script on your local host machine. Name the script something like myFirstADC.c.

```
#include <scriptml/ad7991.h>
#include <scriptml/i2c.h>
#include <stdint.h>
#include <test_print.h>
#include <stdio.h>

int main()
{
    sml_ad7991_t adc = sml_ad7991_create( sml_i2c_open( "/dev/i2c-1" ), 0 );
    uint8_t rd_buf[2], wr_buf[1];

    /* Get samples from Vin0 and Vin1 alternately, all other configs left at
       defaults */
    wr_buf[0] = 0x30;

    /* Write the config value to the ADC */
    sml_ad7991_write( adc, wr_buf, 0, 1 );

    sleep(1);

    int i;
```

```
/* Define two registers to hold the 16 bit ADC register contents. The 12
 * LSB are
 * the ADC value, and the two bits above that, the channel identity */
uint8_t MSB, LSB;
uint8_t ChannelNumber;

int adcValue;

/* Read contents of ADC */
while(1){

    /* Read from the ADC, two bytes */
    sml_ad7991_read( adc, rd_buf, 0, 2 );
    MSB = rd_buf[0];
    LSB = rd_buf[1];

    /* Mask and bit shift to get the channel identifier */
    ChannelNumber = (MSB & 0x30) >> 4;

    /* Get the 12 bit ADC value */
    adcValue = LSB + ((MSB & 0x0F) << 8);

    /* Print the values */
    printf("Channel = %d, Value = %d\n", ChannelNumber, adcValue);

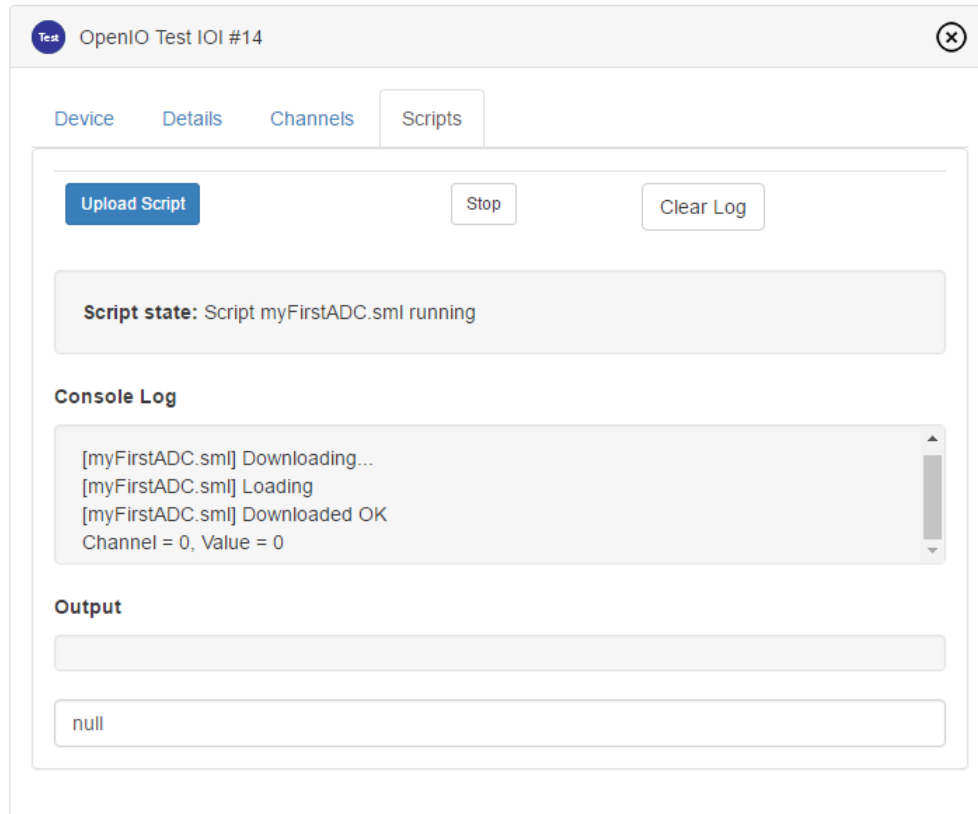
    /* Pause for next sample */
    sleep(1);
}

return 0;
}
```

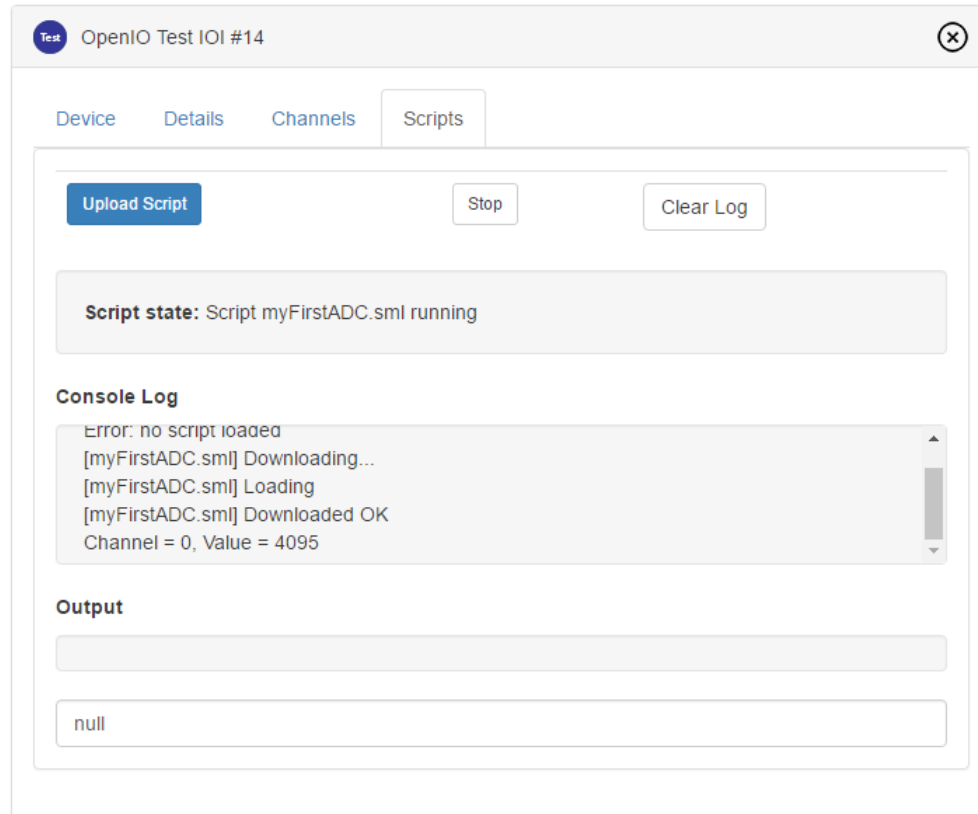
Example ADC Driver written in C

To test the ADC and the script, connect the ADC input to the Ground rail of the device (0 Volts).

When you are ready, upload the script as before and click on Start. You will see the following window. At the bottom of the window, you can see the printout from the Script with the channel number equal to 0 and the ADC value read from the ADC also 0. As the analogue input to the ADC is 0 Volts, the digital value that we read back is 0.



If we now stop the script, and change the input to the ADC from 0 Volts to 5 Volts, which is the same as the power supply for the device. We can then re-run the script and we will see the following in the browser window.



In this instance, we now have a 5 Volt input. As the ADC is a 12 bit device, this corresponds to the maximum value, and hence a digital value of 4095, as seen.

ADC Driver Script in Python

Next we can create a similar driver script, but in this case we will use the Python language. Create a file myFirstADC.py and add the following lines to the file:

```
import i2c
import ad7991
import time

# Instantiate the device
adc = ad7991.device( i2c.device("/dev/i2c-1"), 0)

# Set the number of ADC channels
adc.set_channel(ad7991.TWO_CHAN)

# Get the value for channel 0
channel, value = adc.get_level()
print ('Channel = ' + channel + ' ADC value = ' + value)

# Get the value for channel 1
channel, value = adc.get_level()
```

```
print ('Channel = ' + channel + ' ADC value = ' + value)

# Destroy the device
del adc
```

Example ADC Driver written in Python

As before, if we load the script using the browser, then run the script, we will see the same outputs as we obtained with the C script. Try adjusting the voltage input to the ADC and see the effects that we get on the values that are measured by the ADC and displayed on the browser.

Chapter 4: Connecting to the Server

Now that we have some simple hardware tests working in both C and Python, we can now start to push the data that we collect to the server, and display the results on the browser.

In this example, we will use both an ADC and a DAC. We will be setting the voltage levels for the DAC using a script. The analogue voltage output of the DAC is connected to the analogue input of the ADC. We will then read the ADC value using the script, and then push the result up to the server for data collection and display.

In this example, we will start to use some of the LWM2M objects that are defined by OpenIO Labs to be used in the OpenIO Labs system. For an introduction to the LWM2M objects, the LWM2M objects created by OpenIO Labs, and their use with ScriptML, refer to the Ref [8] User Guide which goes into the details of the LWM2M objects and how they are used within ScriptML.

As before we will create both a C and a Python version of these scripts and test each in turn.

ADC and DAC Driver and Connecting to the Server – in C

We will start by introducing the LWM2M OpenIO Labs Motor Control Object. This object was created using the identical principles of the LWM2M objects, registered in the User-Defined object space defined by LWM2M and allows the user to control a motor using a device such as a DAC to produce a variable voltage.

In this example, we will not be actually using a motor, but rather simulating the drive to the motor and recording the motor drive voltage to the server. In this example we will be using a MAX517 DAC that we saw earlier, with the analogue output connected to the AD7991 ADC. The script that we will write will drive the DAC voltage, and then record the ADC values, and then push these to the server where we can collect the data and present the results.

We will start by creating a file, motorMonitor.c as show in the code presented below.

```
#include <scriptml/ad7991.h>
#include <scriptml/max518.h>
#include <lwm2m.h>

#include <scriptml/i2c.h>
#include <stdint.h>
#include <test_print.h>
#include <stdio.h>

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

int main()
{
    /* Instantiate the ADC and the DAC on the I2C bus */
```

```

sml_ad7991_t adc = sml_ad7991_create( sml_i2c_open( "/dev/i2c-1" ), 0 );
sml_max518_t dac = sml_max518_create( sml_i2c_open( "/dev/i2c-1" ), 0 );

/* Register our object with server.
 * Once done, the LWM2M_OpenIO_Motor_Control is now externally readable
 * and modifiable */
sml_lwm2m_register_object(LWM2M_OpenIO_Motor_Control);

uint8_t rd_buf[2], wr_buf[1];

/* Get samples from Vin0 and Vin1 alternately, all other configs left at defaults */
wr_buf[0] = 0x30;

/* Write the config value to the ADC */
sml_ad7991_write( adc, wr_buf, 0, 1 );

sleep(1);

int iLoop = 0;

/* Define two registers to hold the 16 bit ADC register contents. The 12 LSB are
the ADC value, and the two bits above that, the channel identity */
uint8_t MSB, LSB;
uint8_t ChannelNumber;

int adcValue;

/* Set the DAC vale and then read contents of ADC */
while(1){

    /* Set the DAC level for channels 1 and 2 */
    sml_max518_set_level( dac, 0, iLoop );
    sml_max518_set_level( dac, 1, iLoop );

    /* Pause to allow DAC to settle (much longer than needed here to also slow display) */
    sleep(1);

    /* Read from the ADC, two bytes */
    sml_ad7991_read( adc, rd_buf, 0, 2 );
    MSB = rd_buf[0];
    LSB = rd_buf[1];

    /* Mask and bit shift to get the channel identifier */
    ChannelNumber = (MSB & 0x30) >> 4;

    /* Get the 12 bit ADC value */
    adcValue = LSB + ((MSB & 0x0F) << 8);

    /* Print the values */
    printf("Channel = %d, Value = %d, DAC0 value = %d, DAC1 value = %d\n", ChannelNumber,
        adcValue, iLoop, iLoop);

    if( LWM2M_OpenIO_Motor_Control.State )
    {
        LWM2M_OpenIO_Motor_Control.Velocity = adcValue;
    }

    /* increment the DAC set value for the next iteration */
    iLoop++;
}

return 0;

```

```
}
```

Example ADC and DAC Driver written in C and Connecting to the OpenIO Labs Server

The C code that is presented above is very similar to the code that we used for the DAC and ADC examples that we have done previously. Rather than looking at all of the code, we will focus on the parts that are new, namely the interactions between the ScriptML script and the OpenIO Labs server.

The IOI device that we are using is connected to the OpenIO Labs server using the LWM2M protocol described in Ref [5]. The LWM2M protocol defines a number of different object types that are defined by the standards groups involved. The User Guide (UG404) referenced in Ref [8] provides detailed information for the LWM2M objects that are supported in the OpenIO Labs ScriptML system.

In addition to the standard objects defined by the LWM2M standard, there are also a number of objects that have been defined by OpenIO Labs. The definition of these objects is allowed within the LWM2M standard because the vendor specific object identifier space is used.

In the example that we have here, we will be using the Motor Control object that is defined by OpenIO Labs.

So, exploring the code, we will see the following around line 10 of the code:

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

Here, we are defining an instance of the OpenIO Labs Motor Control object that we will use later.

In the next part of the code, we are creating an instance of the ADC and DAC I2C devices that we will be used from within the script.

Then, in the line of code around line 21, we see:

```
/* Register our object with server.  
 * Once done, the LWM2M_OpenIO_Motor_Control is now externally readable  
 * and modifiable */  
sml_lwm2m_register_object(LWM2M_OpenIO_Motor_Control);
```

Here, we are now registering the motor control object that we have created an instance of, and registered it with the OpenIO Labs LWM2M server. Once this is done, we will be able to interact with the motor control object from within ScriptML.

In the next main code segments, we are setting up the ADC and the DAC. We then enter an infinite loop in which we increment a variable, set the DAC value to that variable using the I2C interface. As the analogue output of the DAC is connected to the analogue input of the ADC, we can read the ADC value.

The DAC is 8 bits, going from 0 to 255, using a 5 Volts supply this corresponds to a voltage range

of 0 Volts to 5 Volts. The ADC is a 12 bits converter that is also connected to a 5 Volts supply, and consequently will read the values of 0 for 0 Volts, and 4095 for 5 Volts.

The value of the ADC is read. As it is 12 bits, we will need to read two registers from the ADC using the I2C protocol, and then convert these two 8 bit numbers into a single 12 bit number.

Finally, we can set the value of the motor control velocity to the ADC value. Note this is a synthetic example, and in practice we would not connect the circuit together in this manner.

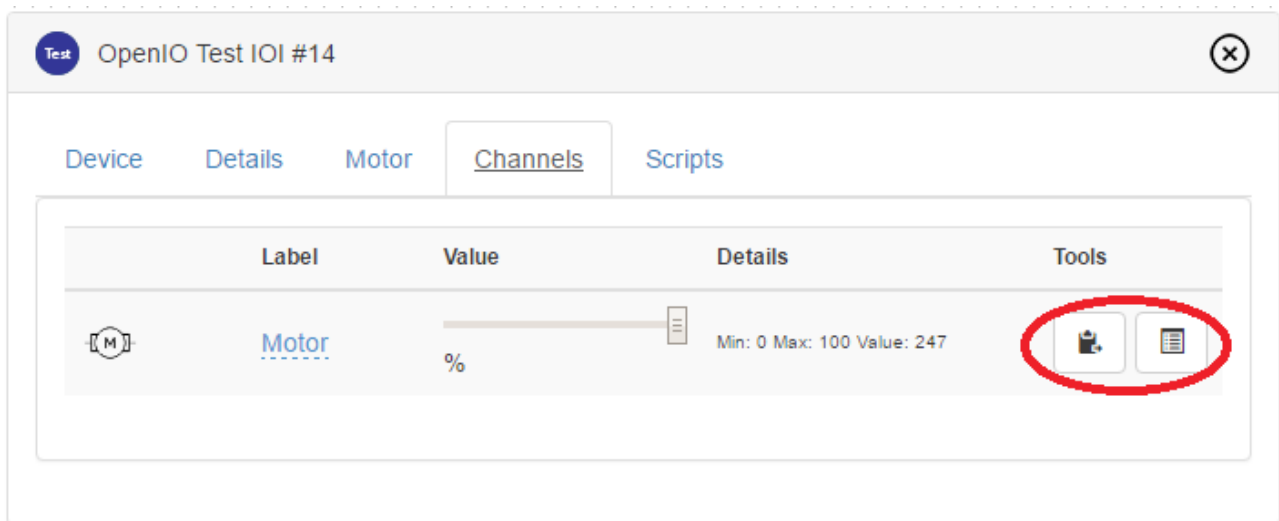
So in the line:

```
LWM2M_OpenIO_Motor_Control.Velocity = adcValue;
```

We are setting the motor control velocity to the output value of the ADC. This is being pushed to the server so that we can record and view the drive values on the server.

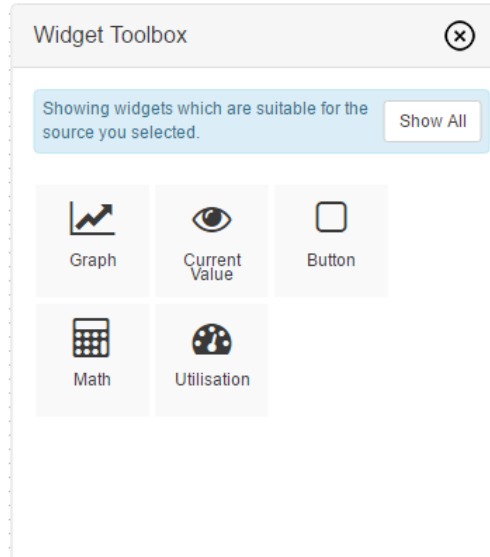
Having completed the C script, we can now upload the script to the server as before. Next, we start the script as previously.

In the next step, we will select the Channels tab as shown below:



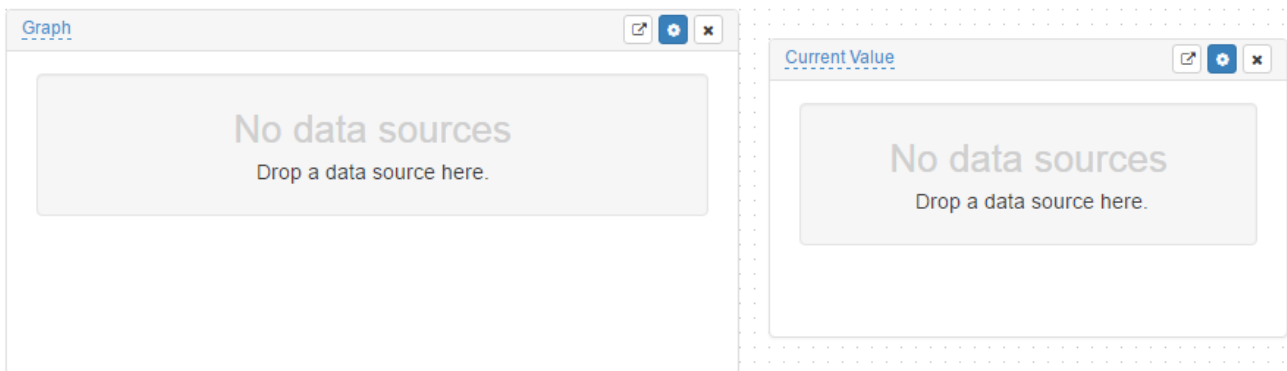
On the right hand side of the channels tab we see two icons. The first icon is the data source icon, and the second icon is the widget icon.

First click on the widget icon, and we will see the widget tool box:



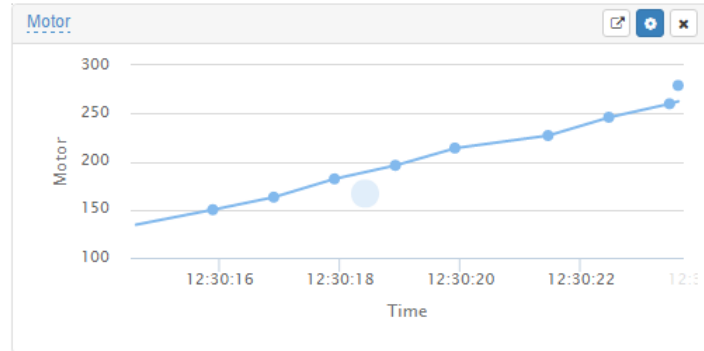
We are going to use a Graph widget and a Current Value widget.

Click on both the Graph Widget and the Current Value Widget and they will appear in the browser as follows:

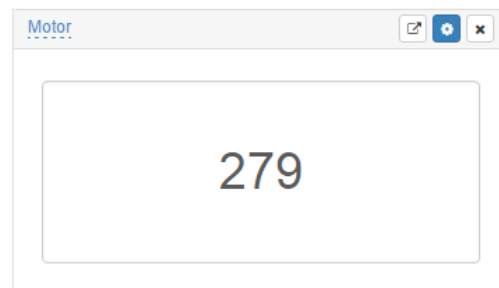


Next we will “grab” the data source icon we saw on the channels tab and drop it first into the Graph widget and then repeat and drop it into the Current Value widget.

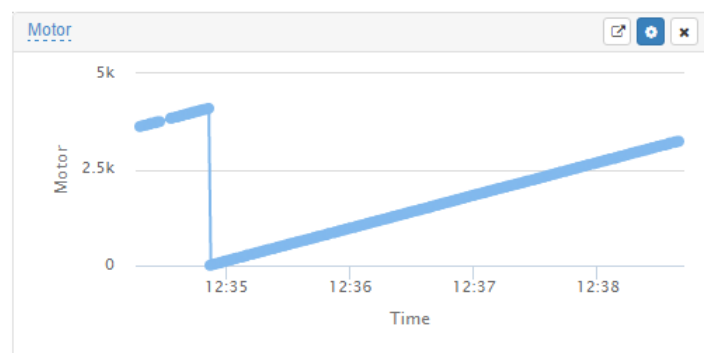
For the Graph widget we will see, something like this:



And for the Current Value, something like this:



For the Graph widget, we can change the range of the axis by clicking on the “settings” icon, which is the middle icon in the top row of the Graph widget. If we do this and change the value from 10 seconds to say, 1000 seconds, click on the green Tick icon, and let the graph run for a few minutes, we will see the ADC output value that looks something like this:



ADC & DAC Driver Connecting to the Server – in Python

Next we will implement a Python script using a similar methodology as the C script. The registration and setting of the motor object is similar to the C case, but with a slightly different syntax as illustrated in the code listing presented below.

```
import i2c
import ad7991
import max518
import time
import lwm2m

# Instantiate the motor, ADC and DAC device
adc = ad7991.device( i2c.device("/dev/i2c-1"), 0)
dac = max518.device( i2c.device("/dev/i2c-1"), 0 )
my_motor = lwm2m.create_OpenIO_Motor_Control()

# Register the motor with the server
# my_motor is now externally readable and modifiable
lwm2m.register_object_id(lambda: my_motor, lwm2m.id_OpenIO_Motor_Control)

# Set the number of ADC channels
adc.set_channel(ad7991.TWO_CHAN)

iLoop = 0;

while True:

    # Set the DAC levels in opposite directions
    dac.set_level( 0, iLoop )
    dac.set_level( 1, 255- iLoop )

    # Allow time for the loop
    time.sleep(1)

    # Get the value for channel 0
    channel, value = adc.get_level()
    print ( 'Channel = ' + channel + ' ADC value = ' + value + ' DAC value = ' + iLoop)

    # Get the value for channel 1
    channel, value = adc.get_level()
    print ( 'Channel = ' + channel + ' ADC value = ' + value + ' DAC value = ' + str(255-iLoop))

    # If the motor is active update its velocity with the ADC value
    if my_motor[lwm2m.OpenIO_Motor_Control_State]:

        my_motor[lwm2m.OpenIO_Motor_Control_Velocity] = value

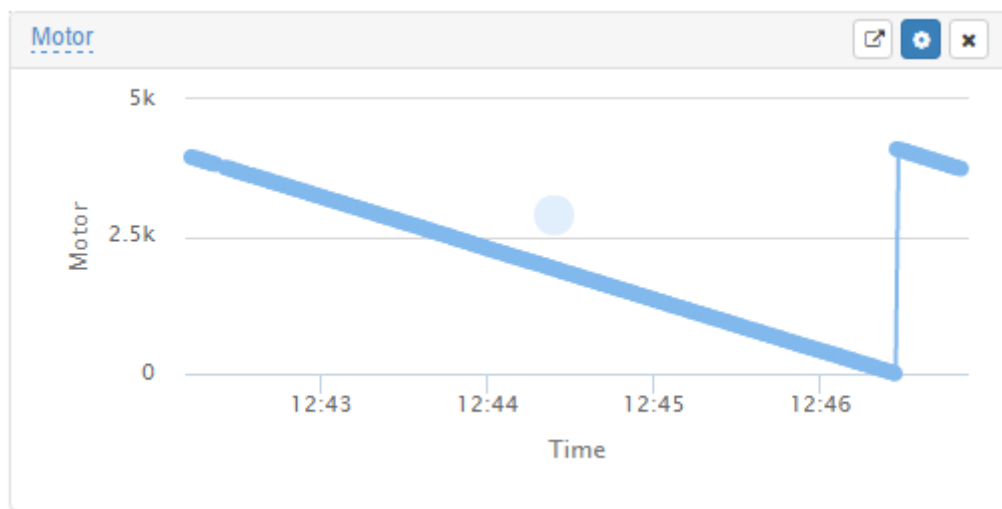
    # Increment DAC value
    iLoop = iLoop + 1

# Destroy the devices
del adc
del dac
del my_motor
```

Example ADC and DAC Driver written in Python and Connecting to the OpenIO Labs Server

In the Python example, the code is slightly different, in that we are reading values from channel 0 and channel 1, but we are pushing the output from channel 1 to the server. In this case, we will see the ramp waveform decrementing, unlike the C case where we were sampling channel 0 with an incrementing ramp.

The waveform that we observe after a few minutes looks like:



Summary

So, to summarise, we have done the following:

1. Create a C and Python script in an editor
2. Added an ADC and DAC to the C or Python source files
3. Added a LWM2M motor control object to the C or Python source files
4. Powered on the IOI device which registered with the server using LWM2M
5. Upload the script to the server and then down to the device
6. Execute the script
7. Instantiate a Graph widget and a Current Value widget
8. Captured, stored and displayed the output of the ADC whilst being driven by a DAC. All of the control code and data collection code implemented in either the C or Python source files.

Conclusions

In this User Guide we have stepped through the stages you will go through to create both simple and complex C or Python scripts that can interface directly with both the OpenIO Labs server and the OpenIO Labs edge devices (IOI).

The IOI and the OpenIO Labs server are inter-connected by the industry standard LWM2M protocol.

User control of both the edge device and the data that is sent to the server can be written in a variety of languages such as C, Python, C++, Java and Golang. Scripts written in these languages are then converted to ScriptML and passed to edge device for execution.

Full access to the LWM2M objects is supported in both the OpenIO Labs server and the edge device through the use of the ScriptML protocol.

We have seen a number of examples written in both C and Python that control a number of devices such as an ADC and DAC.

The use of the OpenIO Labs system can be expanded with the addition of new devices that can be attached to the OpenIO Labs edge device with the user creating scripts to setup and control the device as well as send data from the device to the server for storage or presentation.

Access to the OpenIO Labs server is via an industry standard browser such as Google Chrome.

With such power and flexibility we have seen how simple it is to create a relatively complex system with very simple scripts in both C and Python.

Appendix

C – Header and Python Package Files

The examples in the User Guide use a number of header files and python packages. The details of these files are given here in the appendix.

MAX518.h Header File

The examples in the text that use the DAC pull in the MAX518.h header file. The header file defines the methods that the main script uses to configure and control the MAX518 DAC. The header file can be written by the user and is tailored to the specific device in question.

There are at least two ways to create the header file. In the first instance a sample header file can be sourced using a google search. Most I2C devices have a Linux driver which can be tailored for use with ScriptML. In the second method, a review of the device data sheet can identify the methods that are required.

Ref [7] provides a detailed overview of writing I2C device drivers within ScriptML. That user guide also includes a detailed overview of the i2c.h header file and how it interacts via ScriptML with the underlying hardware layer used by the processor.

For the MAX518.h header file presented below, four methods are defined:

1. Create an instance of the device for use within ScriptML
2. Set the DAC output level for the device
3. Change the device state from off to on
4. Reset the device

All of the methods use the ScriptML I2C library (i2c.h) that allows ScriptML access to the underlying I2C IO interface with a minimal of user-configuration. The I2C library provides simple read and write functions to the I2C bus. Using simple read/write operations, control of both simple and complex I2C devices can be easily achieved using ScriptML.

```
/** @file
 *
 * @brief MAX518 DAC device driver.
 *
 * Support for output level setting on both channels, power down and reset
 *
 * Copyright (c) Open IO Labs Ltd 2016
```

```

* All Rights Reserved
*/

#include <scriptml/i2c.h>

#define BASE_ADDR 0x2C

typedef struct
{
    sml_i2c_t *piic;
    int addr;
} sml_max518_t;

/** @brief Create device driver instance
 *
 * @param i2c An instance of the I2C driver for the bus on which the device is placed
 * @param addr_offset The I2C address offset, corresponding to the value on the Ax pins, 0 through
3 inclusive
 * @return Device handle
 */
sml_max518_t sml_max518_create( sml_i2c_t i2c, int addr_offset )
{
    sml_max518_t dev;
    dev.piic = &i2c;
    dev.addr = BASE_ADDR | addr_offset;
    return dev;
}

/** @brief Set the level of output pins on the IO expander
 *
 * @param dev Device handle
 * @param channel Channel number to set; 0 or 1
 * @param level A byte containing the level; 0 through 255 inclusive
 */
void sml_max518_set_level( sml_max518_t dev, int channel, int level )
{
    struct i2c_msg msg;
    msg.addr = dev.addr;
    msg.flags = 0;
    char b[2];
    msg.buf = b;
    int i=0;
    msg.buf[i++] = channel;
    msg.buf[i++] = level;
    msg.len = i;
    sml_i2c_transfer(*(dev.piic), &msg, 1);
}

/** @brief Set power status
 *
 * @param dev Device handle
 * @param power Power status; 0 for off, 1 for on
 *
 * Note: device is initially powered on
 */
void sml_max518_power( sml_max518_t dev, int power )
{
    struct i2c_msg msg;
    msg.addr = dev.addr;
    msg.flags = 0;
    char b[2];

```



```
        msg.buf = b;
        int i=0;
        msg.buf[i++] = power ? 0x0 : 0x8;
        msg.len = i;
        sml_i2c_transfer(*(dev.piic), &msg, 1);
    }

/** @brief Reset device
 *
 * @param dev Device handle
 *
 * Device is reset as described in the data sheet
 */
void sml_max518_reset( sml_max518_t dev )
{
    struct i2c_msg msg;
    msg.addr = dev.addr;
    msg.flags = 0;
    char b[2];
    msg.buf = b;
    int i=0;
    msg.buf[i++] = 0x10;
    msg.len = i;
    sml_i2c_transfer(*(dev.piic), &msg, 1);
}
```

MAX518.h Header File

MAX518.py Package File

Some examples in the User Guide that use the MAX518 DAC device, pull in the MAX518.py package file. The package defines the methods that the main script uses to configure and control the MAX518 DAC. The file can be written by the user and is tailored to the specific device in question.

There are at least two ways to create the header file. In the first instance a sample header file can be sourced using a google search. Most I2C devices have a Linux driver which can be tailored for use with ScriptML. In the second method, a review of the device data sheet can identify the methods that are required.

For the MAX518.py package presented below, four methods are defined:

1. Create an instance of the device for use within ScriptML
2. Set the DAC output level for the device
3. Change the device state from off to on
4. Reset the device

All of the methods use the Python ScriptML I2C library (i2c.py) that allows ScriptML access to the underlying I2C IO interface with a minimal of user-configuration. The I2C library provides simple read and write functions to the I2C bus. Using simple read/write operations, control of both simple and complex I2C devices can be easily achieved using ScriptML.

```

## @file
# @package max518
#
# @brief MAX518 DAC device driver.
#
# Support for output level setting on both channels, power down and reset
#
# Copyright (c) Open IO Labs Ltd 2016
# All Rights Reserved
#

from numpy import *

BASE_ADDR = 0x2C

## @brief MAX518 device driver class
#
# Instantiate one of these for each device
class device:
    ## @brief Create device driver instance
    #
    # @param i2c An instance of the I2C driver for the bus on which the device is placed
    # @param addr_offset The I2C address offset, corresponding to the value on the Ax pins,
    # 0 through 3 inclusive
    def __init__( self, i2c, addr_offset ):
        self.iic = i2c
        self.addr = BASE_ADDR | addr_offset

    ## @brief Destroy device driver instance
    def __del__( self ):
        del self.iic

    ## @brief Set the level of output pins on the DAC
    #
    # @param channel Channel number to set; 0 or 1
    # @param level A byte containing the level; 0 through 255 inclusive
    def set_level( self, channel, level ):
        command = channel
        m = self.iic.msg( self.addr, 0, [uint8(command), uint8(level)] )
        self.iic.transfer([m])

    ## @brief Set power status
    #
    # @param power Power status; 0 for off, 1 for on
    #
    # Note: device is initially powered on
    def power( self, power ):
        command = 0x0 if power else 0x8
        m = self.iic.msg( self.addr, 0, [uint8(command)] )
        self.iic.transfer([m])

    ## @brief Reset device
    #
    # Device is reset as described in the data sheet
    def reset( self ):
        command = 0x10
        m = self.iic.msg( self.addr, 0, [uint8(command)] )
        self.iic.transfer([m])

```

MAX518.py Package File

AD7991.h Header File

The C examples in the text that use the AD7991 ADC device pull in the AD7991.h header file. The header file defines the methods that the main script uses to configure and control the AD7991 ADC. The header file can be written by the user and is tailored to the specific device in question.

For the AD7991.h header file presented below, three methods are defined:

1. Create an instance of the device for use within ScriptML
2. Write to the ADC for configuration
3. Read from the values from the ADC

All of the methods use the ScriptML I2C library (i2c.h) that allows ScriptML access to the underlying I2C IO interface with a minimal of user-configuration. The I2C library provides simple read and write functions to the I2C bus. Using simple read/write operations, control of both simple and complex I2C devices can be easily achieved using ScriptML

```
/** @file
 *
 * @brief AD7991 8 bit Analogue to Digital Converter (ADC)
 *
 * Read Write support.
 * The write is used to configure the ADC operational modes only
 *
 * Copyright (c) Open IO Labs Ltd 2016
 * All Rights Reserved
 */

#include <scriptml/i2c.h>
#include <stdint.h>

/* Note, this is the address for the AD7991-0 there are other
 * devices in the family with different addresses */
#define BASE_ADDR 0x28

/* Driver flags for this device:      { "AD7991",      ADC }, */

typedef struct
{
    sml_i2c_t *piic;
    int addr;
} sml_ad7991_t;

/** @brief Create device driver instance
 *
 * @param i2c An instance of the I2C driver for the bus on which the device is
 * placed
 * @param addr_offset - not needed for the ADC.
 * @return Device handle
 */
sml_ad7991_t sml_ad7991_create( sml_i2c_t i2c, int addr_offset )
{
    sml_ad7991_t dev;
    dev.piic = &i2c;
```

```
dev.addr = BASE_ADDR | addr_offset;
return dev;
}

/** @brief Write to AD7991 ADC
 *
 * @param dev Device handle
 * @param buf Buffer of data to write
 * @param offset - Not used set to 0
 * @param count - Number of bytes to write
 */
void sml_ad7991_write( sml_ad7991_t dev, const uint8_t *buf,
                      unsigned int offset, size_t count )
{
    struct i2c_msg msg[1];

    msg[0].addr = dev.addr;
    msg[0].flags = 0;
    msg[0].buf = buf;
    msg[0].len = count;

    sml_i2c_transfer(*(dev.piic), msg, 1);
}

/** @brief Read from AD7991 ADC
 *
 * @param dev Device handle
 * @param buf Buffer of data for read
 * @param offset - Not used set to 0
 * @param count - Number of bytes to read
 */
void sml_ad7991_read( sml_ad7991_t dev, uint8_t *buf,
                     unsigned int offset, size_t count )
{
    struct i2c_msg msg[1];

    msg[0].addr = dev.addr;
    msg[0].flags = I2C_M_RD;
    msg[0].buf = buf;
    msg[0].len = count;

    sml_i2c_transfer(*(dev.piic), msg, 1);
}
```

AD7991.h Header File

AD7991.py Package File

The Python examples in the text that use the AD7991 ADC device pull in the AD7991.py package file. The package defines the methods that the main script uses to configure and control the AD7991 ADC. The package file can be written by the user and is tailored to the specific device in question.

For the AD7991.py header file presented below, three methods are defined:

1. Create an instance of the device for use within ScriptML
2. Write to the ADC for configuration

3. Read from the values from the ADC

```
## @file
# @package ad7991
#
# @brief AD7991 ADC device driver.
#
# Support for ADC operations
#
# Copyright (c) Open IO Labs Ltd 2016
# All Rights Reserved
#

from numpy import *
import i2c

BASE_ADDR = 0x28

ONE_CHAN    = 0x10
TWO_CHAN    = 0x30
THREE_CHAN  = 0x70
FOUR_CHAN   = 0xF0

## @brief AD7991 device driver class
#
# Instantiate one of these for each device
class device:
    ## @brief Create device driver instance
    #
    # @param i2c An instance of the I2C driver for the bus on which the device is placed
    # @param addr_offset The I2C address offset, corresponding to the value on the Ax pins
    # @param numChannel One of ONE_CHAN, TWO_CHAN, THREE_CHAN, FOUR_CHAN
    def __init__( self, i2c, addr_offset ):
        self.iic = i2c
        self.addr = BASE_ADDR | addr_offset

    ## @brief Destroy device driver instance
    def __del__( self ):
        del self.iic

    def set_channel( self, channel):
        self.control = channel
        c = self.iic.msg( self.addr, 0, [uint8(self.control)] )
        self.iic.transfer([c])

    ## @brief Get the level of the analogue inputs, the device will cycle through all active inputs
    #
    # @param channel Channel number to read; 0 thru 3 depending on mode
    # @return A byte containing the read back value
    #
    # Note: the channel numbers given here map to the AINx pins in mode
    # FOUR_SINGLE. For other modes, see the data sheet.
    #
    def get_level( self ):
        MSB = uint8(0)
        LSB = uint8(0)
        r = self.iic.msg( self.addr, i2c.I2C_M_RD, [MSB, LSB] )
        self.iic.transfer([r])
        # The returned value will include the channel identity and the 12 bit value
```

```
channel = (MSB & 0x30)>> 4  
value = LSB + ((MSB & 0x0F) << 8)  
return channel, value
```

AD7991.py Package File

Additional Resources

References

The following documents and references are cited within this guide:

1. UG101 – OpenIO Labs System Architecture
2. UG102 – Accessing the OpenIO Labs System
3. UG401 – ScriptML System Overview
4. <https://en.wikipedia.org/wiki/I2C>
5. https://en.wikipedia.org/wiki/OMA_LWM2M
6. <https://www.maximintegrated.com/en/products/analog/data-converters/digital-to-analog-converters/MAX518.html>
7. UG403 – Writing I2C Device Drivers in C and Python
8. UG404 – LWM2M Object Support in ScriptML
9. http://www.analog.com/media/en/technical-documentation/data-sheets/AD7991_7995_7999.pdf