Method SCRIPT SDK Example





Last revision: March ? 2019

© 2019 PalmSens BV

www.palmsens.com

## Contents:

The arduino example “MethodScriptExample.ino” found in the “/MethodScriptExample” folder demonstrates basic communication with the EmStat Pico through Arduino MKR ZERO using the embedded SDK (C libraries). The example allows the user to start measurements on the EmStat Pico from the PC connected to the Arduino through USB.

## Hardware setup:

* To run this example, connect your Arduino MKRZERO "Serial1" port Rx (pin 13), Tx (pin 14) and GND to the EmStat Pico “Serial” Tx, Rx and GND respectively.
* Make sure the UART switch block SW4 on the EmStat dev board has the switches for MKR 3 and 4 turned on.
* The Arduino board should be connected normally to a PC.
* If not powering the EmStat by other means, the EmStat Pico should be connected to the PC through USB for power.

## Environment setup:

* To run this example, you must include the Method SCRIPT C libraries first.
* To do this, follow the menu "Sketch -> Include Library -> Add .ZIP/Library..." and select the MethodScriptCommfolder.

## Communications

In order to use the C library, MSComm, the “extern C” wrapper has to be used because Arduino uses a C++ compiler.

extern "C" {

#include <MSComm.h>

#include <MathHelpers.C>

};

As MSComm is the communication object with the EmStat Pico it needs some read/write functions to be passed in through the MSCommInit(in MSCommInit). However, because the C compiler doesn't understand C++ classes,

The write/read functions from the Serial class are wrapped in a normal function, first.

int write\_wrapper(char c)

{

if(\_printSent == true)

{

//Send all data to PC as well for debugging purposes

Serial.write(c);

}

return Serial1.write(c);

}

int read\_wrapper()

{

int c = Serial1.read();

if(\_printReceived == true && c != -1) //-1 means no data

{

//Send all received data to PC for debugging purposes

Serial.write(c);

}

return c;

}

A new UART instance is then created and assigned to TX (14) and RX (13) pins on the Arduino.

Uart Serial1(&sercom5, 14, 13, SERCOM\_RX\_PAD\_3, UART\_TX\_PAD\_2);

The interrupt handler is attached to the SERCOM.

void SERCOM5\_Handler()

{

Serial1.IrqHandler();

}

### Connecting to the device

The code within the setup() function is executed only once.

Init the Serial and Serial1 ports in here with the baud rate of the EmStat Pico - 230400

Serial.begin(230400);

Serial1.begin(230400);

Assign the pins 13 and 14 to PIO\_SERCOM\_ALT

//Assign SDA (serial data line) function to pin 13

pinPeripheral(13, PIO\_SERCOM\_ALT);

//Assign SCL (serial clock line) function to pin 14

pinPeripheral(14, PIO\_SERCOM\_ALT);

The read/write wrapper functions and a new instance of MSComm object are passed on to MSCommInit for initialization.

MSComm \_espComm;

RetCode code = MSCommInit(&\_espComm, &write\_wrapper, &read\_wrapper);

### Verifying the connected device

Inorder to verify if the device connected is EmStat Pico, the command to get version string “t\n” is sent to the arduino.

const char\* CMD\_VERSION\_STRING = "t\n";

SendScriptToDevice(Cmd\_versionString);

void SendScriptToDevice(const char\* scriptText)

{

for(int i = 0; i < strlen(scriptText); i++)

{

Serial1.write(scriptText[i]);

}

}

The response from the arduino is is read character by character until the new line character is read to form the version string. If the version string contains the string “esp” it is identified as EmStat Pico.

while (Serial1.available())

{

char incomingByte\_pico = Serial1.read();

\_versionString[i++] = incomingByte\_pico;

if(incomingByte\_pico == '\n')

{

\_versionString[i] = '\0';

break;

}

delay(20);

}

if(strstr(\_versionString, "espbl")

{

Serial.println("EmStat Pico is connected in boot loader mode.");

return 0;

}

else if(strstr(\_versionString, "esp"))

{

Serial.println("Connected to EmStat Pico.");

return 1;

}

else

{

Serial.println("Could not connect to EmStat Pico");

}

}

### Sending the method script

The measurement configuration parameters are stored in a static constant string. For example, in the above example, the measurement configuration parameters are stored in a char array as below.

//LSV measurement configuration parameters

const char\* LSV\_METHOD\_SCRIPT = "e\n"

"var c\n"

"var p\n"

"set\_pgstat\_mode 3\n"

"set\_max\_bandwidth 200\n"

"set\_cr 500u\n"

"set\_e -500m\n"

"cell\_on\n"

"wait 1\n"

"meas\_loop\_lsv p c -500m 500m 50m 100m\n"

"pck\_start\n"

"pck\_add p\n"

"pck\_add c\n"

"pck\_end\n"

"endloop\n"

"cell\_off\n\n";

The measurement configuration parameters are then sent to the arduino.

SendScriptToDevice(LSV\_METHOD\_SCRIPT);

void SendScriptToDevice(const char\* scriptText)

{

for(int i = 0; i < strlen(scriptText); i++)

{

Serial1.write(scriptText[i]);

}

}

### Receiving response

The code to receive and parse the response from the device is written in the loop() so that it runs repeatedly and the response can be ontained from the device as and when it is available.

Inorder to read and parse the response from the device, the Receive Package function from the MSComm library can be used in the arduino code as below.

while (Serial1.available())

{

//Read from the device and try to identify and parse a package

RetCode code = ReceivePackage(&\_espComm, &data);

…

}

The ReceivePackage() function in the C library reads character by character until end of line (‘\n’) from the device using the msComm read wrapper readCharFunc(). The characters are then combined in to a line of response.

**int** tempChar; //Temporary character used for reading

**int** i = 0;

**do** {

tempChar = msComm->readCharFunc();

**if**(tempChar > 0)

{

buf[i++] = tempChar; // Store tempchar into buffer

**if**(buf[0] == (**int**)'e')

**return** *CODE\_RESPONSE\_BEGIN*;

**if**(tempChar == '\n')

{

buf[i] = '\0';

**if**(buf[0] == *REPLY\_MEASURING*)

**return** *CODE\_MEASURING*;

**else** **if**(**strcmp**(buf, "\*\n") == 0)

**return** *CODE\_MEASUREMENT\_DONE*;

**else** **if**(**strcmp**(buf, "\n") == 0)

**return** *CODE\_RESPONSE\_END*;

**else** **if**(buf[0] == *REPLY\_MEASURE\_DP*)

**return** *CODE\_OK*;

**else**

**return** *CODE\_NOT\_IMPLEMENTED*;

}

}

} **while** (i < 99);

buf[i] = '\0';

**return** *CODE\_NULL*;

### Parsing the response

Each line of response returned by the method ReadResponseLine(), can be added to a string to form the raw data if needed. The response line can be further parsed if it is identified to be a data package.

while (true)

{

readLine = ReadResponseLine(); // Read a line from the response

RawData += readLine; // Add the response to raw data

if (readLine == "\n")

break;

else if (readLine.Contains("P"))

{

//Increment the number of data points if the read line contains the header char 'P’

NDataPointsReceived++;

ParsePackageLine(readLine); // Parse the line read

}

}

Here’s a sample response (raw data) from a Linear sweep voltammetric measurement.

eM0000\n

Pda7F85F3Fu;ba48D503Dp,10,288\n

Pda7F9234Bu;ba4E2C324p,10,288\n

Pda806EC24u;baAE16C6Dp,10,288\n

Pda807B031u;baB360495p,10,288\n

\*\n

\n

While parsing the response, various identifiers are used to identify the type of response packages. For example, In the above sample response package,

1. ‘e’ marks the beginning of a response.
2. ‘M’ marks the beginning of a measurement loop.
3. ‘P’ marks the beginning of a row of data package.
4. “\*\n” marks the end of measurement.
5. “\n” marks the end of response.

The following information can be found in the data packages received from the device.

* Potential (set cell potential in V)
* Current (measured current in A)
* In case of Impedance spectroscopy measurements, the following data values can be obtained from the response
* Frequency (set frequency in Hz)
* Real part of complex Impedance (measured impedance Ohm)
* Imaginary part of complex Impedance (measured impedance in Ohm)

The following meta data values if present can also be obtained from the data packages.

* CurrentStatus (OK, underload, overload, overload warning)
* CurrentRange (the current range in use at the moment)
* Noise (Noise)

#### Parsing the parameter values

Each row of data package begins with the header ‘P’ which is removed first.

int startingIndex = packageLine.IndexOf('P');

string responsePackageLine = packageLine.Remove(startingIndex, 1);

The rest of the package is contains the parameters which are then separated by the delimiter character ‘;’

parameters = responsePackageLine.Split(';');

Each of the parameters are then parsed.

The initial two characters of every parameter identifies the parameter.

paramIdentifier = parameter.Substring(startingIndex, 2);

For example, in the sample package seen above, the parameter identifiers are

‘da7F85F3Fu’ - ‘da’ Potential reading and

‘ba48D503Dp,10,288’ – ‘ba’ current reading.

The parameter values hold the next 8 characters.

const int PACKAGE\_PARAM\_VALUE\_LENGTH = 8;

paramValue = responsePackageLine.Substring(startingIndex + 2, PACKAGE\_PARAM\_VALUE\_LENGTH);

The parameter value for current reading (8 characters)from the above sample package is ‘48D503Dp’. This value is further parsed to retrieve the actual parameter value with the respective unit prefix.

double paramValueWithPrefix = ParseParamValues(paramValue);

The SI unit prefix from the package can be obtained from the parameter value at position 8

char strUnitPrefix = paramValueString[7];

In the above sample package, the unit prefix for current data is ‘p’ which is 1e-12 A.

The code below parses the actual parameter value excluding the unit prefix (7 characters) and appends the respective prefixes.

string strvalue = paramValueString.Remove(7);

The value is first converted from hex to int

int value = Convert.ToInt32(strvalue, 16);

Then value is then adjusted with the Offset value to receive only positive values.

const int OFFSET\_VALUE = 0x8000000;

double paramValue = value - OFFSET\_VALUE;

The value of the parameter is returned after appending the SI unit prefix

return (paramValue \* SI\_Prefix\_Factor[strUnitPrefix.ToString()]);

The SI unit prefixes are as follows.

readonly static Dictionary<string, double> SI\_Prefix\_Factor = new Dictionary<string, double>

{ { "a", 1e-18 },

{ "f", 1e-15 },

{ "p", 1e-12 },

{ "n", 1e-9 },

{ "u", 1e-6 },

{ "m", 1e-3 },

{ " ", 1 },

{ "K", 1e3 },

{ "M", 1e6 },

{ "G", 1e9 },

{ "T", 1e12 },

{ "P", 1e15 },

{ "E", 1e18 }};

The parameter values can be added to the corresponding arrays based on the parameter identifiers.

The potential readings are identified by the string “da”

The current readings are identified by the string “ba”

switch (paramIdentifier)

{

case "da": // Potential reading

// If potential reading add the value to the VoltageReadings array

VoltageReadings.Add(paramValueWithPrefix);

break;

case "ba": // Current reading

// If current reading add the value to the CurrentReadings array

CurrentReadings.Add(paramValueWithPrefix);

break;

}

In case of Impedance sprctroscopy measurement, the following identifiers are used.

The frequency readings are identified by the string “dc”

The real impedance readings are identified by the string “cc”

The imaginary impedance readings are identified by the string “cd”

switch (paramIdentifier)

{

case "dc": //Frequency reading

//If frequency reading add the value to the FrequencyReadings list

FrequencyValues.Add(paramValueWithPrefix);

break;

case "cc": //Real Impedance reading

//If Z(Real) reading add the value to RealImpedanceReadings list

RealImpedanceValues.Add(paramValueWithPrefix);

break;

case "cd": //Imaginary Impedance reading

//If Z(Img) reading add the value to ImgImpedanceReadings list

ImgImpedanceValues.Add(paramValueWithPrefix);

break;

}

After obtaining the parameter identifier and the parameter values from the package, the meta data values if present can be parsed. Meta data values if present are separated by the demiliter character ‘,’

ParseMetaDataValues(parameter.Substring(10));

#### Parsing the meta data values

The meta data values are separated based on the delimiter ‘,’ and each of the values is further parsed to get the actual value. The first character of each meta data value identifies the type of meta data.

‘1’ – status

‘2’ – Current range index

‘4’ - Noise

The status is 1 character hex bit mask. It is converted to long int.

The status can be obtained as shown in the code below.

string status = "";

long statusBits = (Convert.ToInt32(metaDatastatus[1].ToString(), 16));

0 indicates OK

if ((statusBits & 0x0) == (long) ReadingStatus.OK)

status = "OK";

1 indicates overload

if ((statusBits & 0x2) == (long) ReadingStatus.Overload)

status = "Overload";

4 indicates underload

if ((statusBits & 0x4) == (long) ReadingStatus.Underload)

status = "Underload";

and 8 indicates overload warning (80% of maximum).

if ((statusBits & 0x8) == (long) ReadingStatus.Overload\_Warning)

status = "Overload warning";

For example, in the above sample, the available meta data values for current data are,

10,288. The first meta data value is 10.

1 – meta data status – 0 indicates OK.

The meta data type current range is 2 characters long hex value. If the first bit high (0x80) it indicates a high speed mode current range.

The possible current ranges supported by the EmStat Pico are as in the code below.

public enum CurrentRanges

{

cr100nA = 0,

cr2uA = 1,

cr4uA = 2,

cr8uA = 3,

cr16uA = 4,

cr32uA = 5,

cr63uA = 6,

cr125uA = 7,

cr250uA = 8,

cr500uA = 9,

cr1mA = 10,

cr5mA = 11,

//High speed mode current ranges

hscr100nA = 128,

hscr1uA = 129,

hscr6uA = 130,

hscr13uA = 131,

hscr25uA = 132,

hscr50uA = 133,

hscr100uA = 134,

hscr200uA = 135,

hscr1mA = 136,

hscr5mA = 137,

}

The code below can be used to get current range from the package.

string currentRangeStr = "";

byte crByte;

if(byte.TryParse(metaDataCR.Substring(1,2), NumberStyles.AllowHexSpecifier, CultureInfo.InvariantCulture, out crByte))

{

switch (crByte)

{

case (byte)CurrentRanges.cr100nA:

currentRangeStr = "100nA";

break;

case (byte)CurrentRanges.cr2uA:

currentRangeStr = "2uA";

break;

case (byte)CurrentRanges.cr4uA:

currentRangeStr = "4uA";

break;

case (byte)CurrentRanges.cr8uA:

currentRangeStr = "8uA";

break;

case (byte)CurrentRanges.cr16uA:

currentRangeStr = "16uA";

break;

case (byte)CurrentRanges.cr32uA:

currentRangeStr = "32uA";

break;

case (byte)CurrentRanges.cr63uA:

currentRangeStr = "63uA";

break;

case (byte)CurrentRanges.cr125uA:

currentRangeStr = "125uA";

break;

case (byte)CurrentRanges.cr250uA:

currentRangeStr = "250uA";

break;

case (byte)CurrentRanges.cr500uA:

currentRangeStr = "500uA";

break;

case (byte)CurrentRanges.cr1mA:

currentRangeStr = "1mA";

break;

case (byte)CurrentRanges.cr5mA:

currentRangeStr = "15mA";

break;

case (byte)CurrentRanges.hscr100nA:

currentRangeStr = "100nA (High speed)";

break;

case (byte)CurrentRanges.hscr1uA:

currentRangeStr = "1uA (High speed)";

break;

case (byte)CurrentRanges.hscr6uA:

currentRangeStr = "6uA (High speed)";

break;

case (byte)CurrentRanges.hscr13uA:

currentRangeStr = "13uA (High speed)";

break;

case (byte)CurrentRanges.hscr25uA:

currentRangeStr = "25uA (High speed)";

break;

case (byte)CurrentRanges.hscr50uA:

currentRangeStr = "50uA (High speed)";

break;

case (byte)CurrentRanges.hscr100uA:

currentRangeStr = "100uA (High speed)";

break;

case (byte)CurrentRanges.hscr200uA:

currentRangeStr = "200uA (High speed)";

break;

case (byte)CurrentRanges.hscr1mA:

currentRangeStr = "1mA (High speed)";

break;

case (byte)CurrentRanges.hscr5mA:

currentRangeStr = "5mA (High speed)";

break;

}

For example, in the above sample, the second meta data available is 288.

2 – indicates the type – current range

88 – indicates the hex value for current range index – 1mA. The first bit 8 implies that it is high speed mode current range.

### Plotting the response

The OxyPlot library from NuGet packages is used in this example for showing a simple plot of the measurement response parameters.

using OxyPlot;

using OxyPlot.Axes;

using OxyPlot.Series;

A new instance of PlotModel is created and set as the model to the plot object created in the UI.

private PlotModel plotModel = new PlotModel();

samplePlotView.Model = plotModel;

A new instance of LineSeries is also created and added to the plot model.

private LineSeries plotData;

plotData = new LineSeries()

{

Color = OxyColors.Green,

MarkerType = MarkerType.Circle,

MarkerSize = 6,

MarkerStroke = OxyColors.White,

MarkerFill = OxyColors.Green,

MarkerStrokeThickness = 1.5,

};

plotModel.Series.Add(plotData);

The axes for the plot model can be set as shown in the code below.

private void SetAxes()

{

var xAxis = new LinearAxis()

{

Position = OxyPlot.Axes.AxisPosition.Bottom,

MajorGridlineStyle = LineStyle.Dash,

Title = "Potential (V)"

};

var yAxis = new OxyPlot.Axes.LinearAxis()

{

Position = OxyPlot.Axes.AxisPosition.Left,

MajorGridlineStyle = LineStyle.Dash,

Title = "Current (uA)"

};

//Set the x-axis and y-axis for the plot model

plotModel.Axes.Add(xAxis);

plotModel.Axes.Add(yAxis);

}

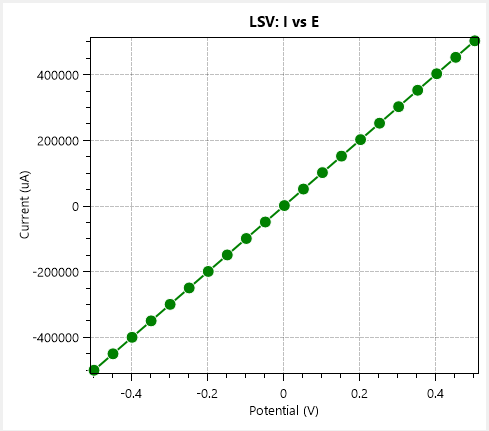
As and when a new line of response is read and parameter values are added to their corresponding arrays, the plot is updated.

// Add the last added measurement values as new data points and update the plot

plotData.Points.Add(new DataPoint(VoltageReadings.Last(), CurrentReadings.Last()));

plotModel.InvalidatePlot(true);

Here’s a sample plot with the response for a Linear sweep measurement on EmStat Pico.



In case of an Impedance spectroscopy measurement, two separate plot objects and plot models are to be created. One for Nyquist plot and the other for Bode plot.

The plot type is set to ‘Cartesian’ type in case of Nyquist plot model and ‘XY’ type in case of Bode plot model.

plotModel.PlotType = plotModel.Equals(NyquistPlotModel) ? PlotType.Cartesian : PlotType.XY;

The plot axes for both the plot models are set as below.

Nyquist plot axes are set to linear axes.

var xAxisNyquistPlot = GetLinearAxis("Z (Ohm)", AxisPosition.Bottom);

var yAxisNyquistPlot = GetLinearAxis("-Z'' (Ohm)", AxisPosition.Left);

//Add the axes to the Nyquist plot model

NyquistPlotModel.Axes.Add(xAxisNyquistPlot);

NyquistPlotModel.Axes.Add(yAxisNyquistPlot);

Bode plot axes are set to logarithmic axes.

var xAxisBodePlot = GetLogAxis("Frequency", "Frequency (HZ)", AxisPosition.Bottom, OxyColors.Black);

var yAxisBodePlot = GetLogAxis("Z", "Z (Ohm)", AxisPosition.Left, OxyColors.Blue);

var yAxisSecondaryBodePlot = GetLogAxis("Phase", "-Phase (deg)", AxisPosition.Right, OxyColors.Red);

//Add the axes to the Bode plot model

BodePlotModel.Axes.Add(xAxisBodePlot);

BodePlotModel.Axes.Add(yAxisBodePlot);

BodePlotModel.Axes.Add(yAxisSecondaryBodePlot);

The magnitude of and phase of impedance is required for the Bode plot. This can be obtained by calculating the complex impedance value the real and imaginary parts of the impedance retrieved from the package as in the code below.

Complex ZComplex = new Complex(RealImpedanceValues.Last(), ImgImpedanceValues.Last());

ComplexImpedanceValues.Add(ZComplex);

ImpedanceMagnitudeValues.Add(ComplexImpedanceValues.Last().Magnitude);

PhaseValues.Add(ComplexImpedanceValues.Last().Phase \* 180 / Math.PI);

The plots are then updated with the data values.

NyquistPlotData.Points.Add(new DataPoint(RealImpedanceValues.Last(), ImgImpedanceValues.Last())); // Add the last added measurement values as new data points and update the plot

NyquistPlotModel.InvalidatePlot(true);

NyquistPlotModel.ResetAllAxes();

BodePlotDataMagnitude.Points.Add(new DataPoint(FrequencyValues.Last(), ImpedanceMagnitudeValues.Last()));

BodePlotDataPhase.Points.Add(new DataPoint(FrequencyValues.Last(), PhaseValues.Last()));BodePlotModel.InvalidatePlot(true);

BodePlotModel.ResetAllAxes();

Here’s are the Nyquist and Bode plots with the response for an Impedance sprctroscopy measurement on EmStat Pico.

