



# LMS API

## *- Quick start guide -*



| | |
|---|---|
| Document version: | 1 |
| Document revision: | 03 |
| Last modified: | 7-28-2017 03:36:40 PM |

# Contents

# Revision History

**Version v01r01**
*Started: 23 Apr, 2016*
Initial version

**Version v01r02**
*Started: 19 Apr, 2017*
Updated LimeSuite installation instructions

**Version v01r03**
*Started: 28 July 2017*
Added source code and detailed descriptions of examples

# 1

# Introduction

The scope of this document is compilation of the LMS API and detailed description of the example applications that utilize LMS API.

# 2

# LMS API

This chapter contains brief description of LMS API.

## 2.1 LMS API compilation

LMS API and examples are part of LimeSuite software. It can be downloaded using git :

```
git clone https://github.com/myriadrf/LimeSuite.git
```

To compile LMS API and examples follow the instructions provided in 'docs/Lime_Suite_Compilation_Guide.pdf.pdf' file. Note that wxWidgets library is not required to compile LMS API and examples.

## 2.2 LMS API function documentation

LMS API function definitions and descriptions can be found in LimeSuite.h file (scr/lime/LimeSuite.h). They are documented using Doxygen comments. Doxygen HTML documentation can also be generated while building LimeSuite. Generating of doxygen documentation is automatically enabled in CMake if doxygen is detected in the system

# 3

# LMS API Examples

This chapter contains description of 3 example applications that demonstrate usage of LMS API. The source code of the example applications can be found in 'src/examples/' directory. After compilation the executable files of the examples should be located in 'build/bin/Release/' directory on Windows systems or in 'build/bin/' directory on Linux systems.

On Linux systems with GNU plot installed, all examples should plot received samples.

The sub-sections of this chapter contain the source code of each example. The source code is broken down to smaller parts and description of each part is provided.

## 3.1 Example 1: basicRX

Demonstrates basic functionality required to receive data from one channel:

- Open device
- Set center frequency
- Set sample rate
- Configure data stream
- Receive samples
- Close device

The example application connects to the first detected LimeSDR device, configures it and receives samples for 5 seconds. The number of samples received per read call is printed while receiving samples.

### 3.1.1 Opening a device

First of all, there is some code before main() function that is common to all examples. At the top of the source file there are some include files. Among them, the file of most interest is 'lime/LimeSuite.h' that provides API for interfacing with LimeSDR. Other includes are standard includes for the console output and timing functions, also header for using GNUPlot in case it is enabled.

```
#include "lime/LimeSuite.h"
#include <iostream>
#include <chrono>
#ifdef USE_GNU_PLOT
#include "gnuPlotPipe.h"
#endif

using namespace std;
```

After includes there is a declaration of device handle (lms_device_t*) that is used by various API calls later in code. Initially, it has to be set to 'NULL' and will be modified when device is opened.

There is also helper function for printing error messages in case some API call fails. It is used in the examples after API function calls to do error reporting. The function calls LMS_GetLastErrorMessage() to get the last error message from LimeSuite libraty and then prints it to the console. After that the device is closed by calling LMS_Close() and the program exits.

```
lms_device_t* device = NULL; //Device structure, should be initialize to NULL

int error()
{
    //print last error message
    cout << "ERROR:" << LMS_GetLastErrorMessage();
    if (device != NULL)
        LMS_Close(device);
    exit(-1);
     }
```

To connect to a device, at first a device list is obtained and then the program connect to one of the devices from the list. The code bellow does the following:

- Creates a device list of size 8 (this should be large enough)
- Populates the device list using LMS_GetDeviceList(). This function returns number of devices found and fills the device list that is passed to it. 'Null' can be passed as device list parameter in order to obtain device count only.
- The number of devices found is printed to the console.
- The first device from the list is opened using LMS_Open().

```
int main(int argc, char** argv)
{
    //Find devices
    int n;
    lms_info_str_t list[8]; //should be large enough to hold all detected devices
    if ((n = LMS_GetDeviceList(list)) < 0) //NULL can be passed to only get number of devices
        error();
    cout << "Devices found: " << n << endl; //print number of devices
    if (n < 1) return -1;

    if (LMS_Open(&device, list[0], NULL))            //open the first device
        error();
```

After connecting to the device, the example loads device with working configuration using LMS_Init(). This is not always necessary. It can be skipped to retain the current configuration of the device or LMS_LoadConfig() can be used to load configuration from a file.

```
//Initialize device with default configuration
//Do not use if you want to keep existing configuration
//Use LMS_LoadConfig(device, "/path/to/file.ini") to load config from INI
if (LMS_Init(device) != 0)
    error();
```

### 3.1.2 Device configuration

The example is using single (first) RX channel. Note that channels are indexed starting from '0' in LMS API. The code snippet bellow does the following:

- Enables the first Rx channel by calling LMS_EnableChannel(). It makes sure that all LMS7 modules required for the first channel are powered up. In this case it is not really necessary as required modules should be enabled after LMS_Init().
- Sets Rx center frequency to 800 MHz using LMS_SetLOFrequency().
- Sets the sample rate to 8 MHz. API function LMS_SetSampleRate() sets sample rate for all channels (Tx and Rx). It also allows to specify oversampling that should be used in hardware. In this case oversampling is 2, which means that hardware will be sampling RF signal at 16 MHz rate and downsampling it to 8 MHz before sending samples to PC.

```
//Enable RX channel
//Channels are numbered starting at 0
if (LMS_EnableChannel(device, LMS_CH_RX, 0, true) != 0)
    error();

//Set center frequency to 800 MHz
if (LMS_SetLOFrequency(device, LMS_CH_RX, 0, 800e6) != 0)
    error();

//Set sample rate to 8 MHz, ask to use 2x oversampling in RF
//This set sampling rate for all channels
if (LMS_SetSampleRate(device, 8e6, 2) != 0)
    error();
```

Before proceeding to streaming setup, the example also enables test signal in Rx using LMS_SetTestSignal(). The example code bellow configures LMS7 to produce NCO generated signal with 8 point constellation in Rx. Note that to receive data from RF, test signal should to be disabled by removing LMS_SetTestSignal() entirely (it is disabled after LMS_Init()) or passing 'LMS_TESTSIG_NONE' instead of 'LMS_TESTSIG_NCODIV8'. The selected antenna port after LMS_Init() should be LNA_H.

```
//Enable test signal generation
//To receive data from RF, remove this line or change signal to LMS_TESTSIG_NONE
if (LMS_SetTestSignal(device, LMS_CH_RX, 0, LMS_TESTSIG_NCODIV8, 0, 0) != 0)
        error();
```

### 3.1.3 Sample streaming setup

Data stream is configured by passing a configuration structure (lms_stream_t) to SetupStream() function. The configuration structure (lms_stream_t) also serves as a stream handle that is used by the other LMS API streaming functions. The example configures the stream as follows:

- Use the first channel (channel=0).
- Set the size of FIFO buffer to ~1 Msample (fifosize = 1024*1024). This sets the size of LMS API buffer for this stream on host PC. The actual size may not be exactly as requested.
- Optimize for a higher data throughput (throughputVsLatency=1.0). This parameter hints whether a lower latency (lower value) or a higher throughput (higher value) is preferred. It affects the size of data transfers from hardware.
- Set up Rx stream (isTx=false),
- Use 12-bit data format (streamId.dataFmt=lms_stream_t::LMS_FMT_I12). Using this format, the samples received from LMS API are 16-bit integer values that range from -2048 to 2047. Data between hardware and PC is transferred using 3 bytes (I(12-bit) +Q(12-bit)) per sample.

```
//Streaming Setup

//Initialize stream
lms_stream_t streamId; //stream structure
streamId.channel = 0; //channel number
streamId.fifoSize = 1024 * 1024; //fifo size in samples
streamId.throughputVsLatency = 1.0; //optimize for max throughput
streamId.isTx = false; //RX channel
streamId.dataFmt = lms_stream_t::LMS_FMT_I12; //12-bit integers
if (LMS_SetupStream(device, &streamId) != 0)
    error();
```

After setting up the stream, basicRX example allocates some buffers for receiving samples and starts streaming by calling LMS_StartStream(). Note that allocated buffers have 16-bit integer data type as it will be receiving 16-bit integers from LMS API. Also, the example is going to read 5000 samples to the buffer at once and one sample has two 16-bit values (I+Q), so buffer that can hold twice as many 16-bit values is required.

```
//Initialize data buffers
const int sampleCnt = 5000; //complex samples per buffer
int16_t buffer[sampleCnt * 2]; //buffer to hold complex values (2*samples))

//Start streaming
LMS_StartStream(&streamId);
```

### 3.1.4 Receiving samples

Receiving samples is done in a while() loop. The loop in basicRX example runs for 5 seconds. The only LMS API functions in the loop is LMS_RecvStream() that read samples from the stream FIFO into the buffer and returns the number of samples read. In this example 'NULL' is passed in place of a metadata structure, therefore timestamps are not obtained. The other code in the loop prints the number of samples read and plots the samples using GNUplot (if enabled.)

```
//Streaming
#ifdef USE_GNU_PLOT
GNUPlotPipe gp;
 gp.write("set size square\n set xrange[-2050:2050]\n set yrange[-2050:2050]\n");
#endif
auto t1 = chrono::high_resolution_clock::now();
while (chrono::high_resolution_clock::now() - t1 < chrono::seconds(5)) //run for 5 seconds
{
      //Receive samples
       int samplesRead; = LMS_RecvStream(&streamId, buffer, sampleCnt, NULL, 1000);
      //I and Q samples are interleaved in buffer: IQIQIQ...
      printf("Received %d samples\n", samplesRead);
          /*
                    INSERT CODE FOR PROCESSING RECEIVED SAMPLES
          */
#ifdef USE_GNU_PLOT
      //Plot samples
      gp.write("plot '-' with points\n");
      for (int j = 0; j < samplesRead; ++j)
           gp.writef("%i %i\n", buffer[2 * j], buffer[2 * j + 1]);
      gp.write("e\n");
      gp.flush();
#endif
}
```

### 3.1.5 Closing the device

At the end of the program streaming is stopped and the device is closed using the following sequence:

- LMS_StopStream() - stops the stream. It does not deallocate the stream so if there is something in the stream FIFO, it can still be read. Also, the stream can be quickly started again using LMS_StartStream().
- LMS_DestroyStream() - deallocates the stream from memory. After this the stream structure can no longer be used.
- LMS_Close() - disconnects from the device and deallocates the device from memory.

```
//Stop streaming
LMS_StopStream(&streamId); //stream is stopped but can be started again with LMS_StartStream()
LMS_DestroyStream(device, &streamId); //stream is deallocated and can no longer be used
//Close device
LMS_Close(device);
return 0;
}
```

### 3.1.6 Application output

The application outputs the number of samples received with each call to LMS_RecvStream() to console (Figure 1). If GNUplot is enabled it also plots constellation of IQ samples (Figure 2).

```
Devices found: 1
[INFO] Estimated reference clock 30.7195 MHz
[INFO] Selected reference clock 30.720 MHz
Received 5000 samples
Received 5000 samples
Received 5000 samples
Received 5000 samples
Received 5000 samples
Received 5000 samples
Received 5000 samples
```

Figure 1: console output of basixRX example

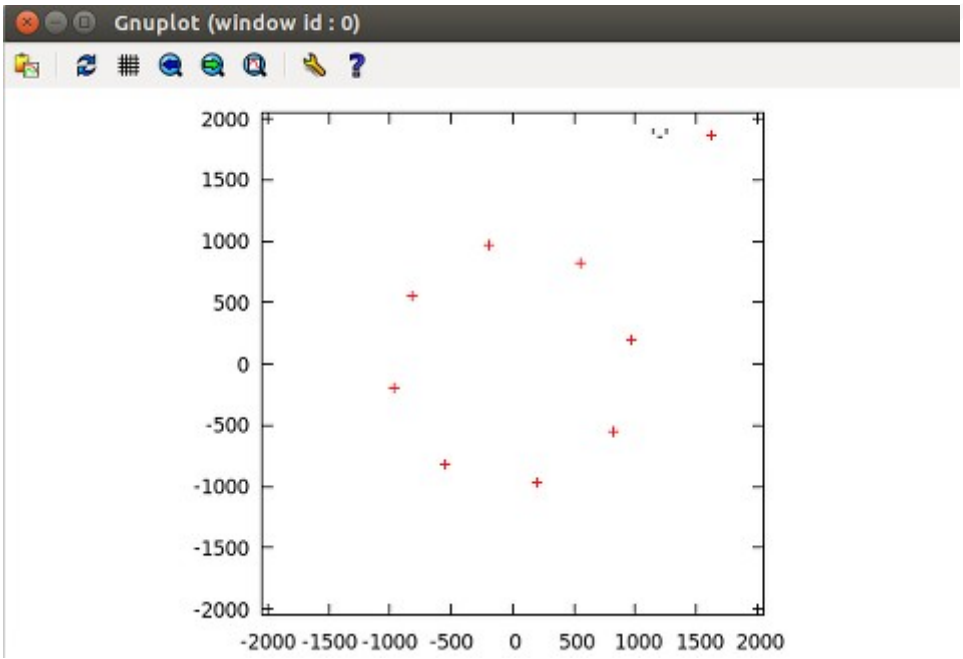


Figure 2: GNUplot output of basicRX example

## 3.2 Example 2: singleRX

More in-depth Rx example than basicRX. Additionally shows how to:

- Obtain the allowed value range from a device
- Obtain currently set device parameters
- Set gains
- Perform auto-calibration
- Set up a low-pass filter
- Select antenna port
- Get a stream status (data rate, FIFO size).

The example application connects to the first detected LimeSDR device, configures it and receives samples for 10 seconds. The data transfer rate and FIFO status is printed every second while streaming is active.

### 3.2.1 Opening a device

The code at the beginning of singleRX examples is the same as in basicRX example, so it is not detailed again in this section.

```
#include "lime/LimeSuite.h"
#include <iostream>
#include <chrono>
#ifdef USE_GNU_PLOT
#include "gnuPlotPipe.h"
#endif

using namespace std;

lms_device_t* device = NULL; //Device structure, should be initialize to NULL

int error() {
   cout << "ERROR:" << LMS_GetLastErrorMessage();    //print last error message
   if (device != NULL)
      LMS_Close(device);
   exit(-1);
}

int main(int argc, char** argv) {
```

This example uses a little different sequence to connect to a device compared to basicRX. It shows how to obtain the number of devices before filling the device list. The code bellow does the following:

- Obtains the number of devices connected to the system by calling LMS_GetDeviceList() and passing 'NULL' as a device list parameter.
- Prints number of devices found to the console.
- Allocates the list based on the number of devices found
- Populates the device list by calling LMS_GetDeviceList()
- Prints the device list to the console
- Opens the first device from the list using LMS_Open().
- Deallocates the device list

```
/Find devices
//First we find number of devices, then allocate large enough list,  and then populate the list
int n;
if ((n = LMS_GetDeviceList(NULL)) < 0)  //Pass NULL to only obtain number of devices
    error();
cout << "Devices found: " << n << endl;
if (n < 1)
    return -1;

lms_info_str_t* list = new lms_info_str_t[n];   //allocate device list
if (LMS_GetDeviceList(list) < 0)              //Populate device list
    error();

for (int i = 0; i < n; i++)                   //print device list
    cout << i << ": " << list[i] << endl;
cout << endl;

//Open the first device
if (LMS_Open(&device, list[0], NULL))
    error();

delete [] list;                           //free device list
```

After connecting to the device, an initial configuration is loaded using LMS_Init() and the first Rx channel is enabled using LMS_EnableChannel().

```
//Initialize device with default configuration
//Do not use if you want to keep existing configuration
//Use LMS_LoadConfig(device, "/path/to/file.ini") to load config from INI
if (LMS_Init(device) != 0)
    error();
//Enable RX channel
//Channels are numbered starting at 0
if (LMS_EnableChannel(device, LMS_CH_RX, 0, true) != 0)
    error();
```

### 3.2.2 Device configuration

In this example it is shown how to set and obtain values of commonly used parameters:

- Center frequency
- Antenna port
- Sample rate
- Analog filter bandwidth
- Gain
- Perform calibration

First of all center frequency is set using LMS_SetLOFrequency() and a read-back is performed using LMS_GetLOFrequency() right after that. Then, the center frequency obtained from the device is printed to the console.

```
//Set center frequency to 800 MHz
if (LMS_SetLOFrequency(device, LMS_CH_RX, 0, 800e6) != 0)
    error();
//print currently set center frequency
float_type freq;
if (LMS_GetLOFrequency(device, LMS_CH_RX, 0, &freq) != 0)
    error();
cout << "\nCenter frequency: " << freq / 1e6 << " MHz\n";
```

The setup continues with Rx RF port selection. The related code that is shown bellow does the following:

- Creates a list for antenna (RF port) names
- Fills the list with the names of Rx RF ports by calling LMS_GetAntennaList()
- Prints the obtained port names to the console
- Obtains the currently set RF port index of the first RX channel using LMS_GetAntenna() and outputs its name to the console.
- Selects RF port for first Rx channel by calling LMS_SetAntenna() and requesting to set RF port to LNAW.
- Obtains the currently set RF port index again and prints its name to the console.

```
   //select antenna port
   lms_name_t antenna_list[10];    //large enough list for antenna names.
         //Alternatively, NULL can be passed to LMS_GetAntennaList() to obtain number of antennae

   if ((n = LMS_GetAntennaList(device, LMS_CH_RX, 0, antenna_list)) < 0)
      error();

   cout << "Available antennae:\n";          //print available antennae names
   for (int i = 0; i < n; i++)
      cout << i << ": " << antenna_list[i] << endl;

   if ((n = LMS_GetAntenna(device, LMS_CH_RX, 0)) < 0) //get currently selected antenna index
      error();
   //print antenna index and name
   cout << "Automatically selected antenna: " << n << ": " << antenna_list[n] << endl;

   if (LMS_SetAntenna(device, LMS_CH_RX, 0, LMS_PATH_LNAW) != 0) // manually select antenna
      error();

   if ((n = LMS_GetAntenna(device, LMS_CH_RX, 0)) < 0) //get currently selected antenna index
      error();
  //print antenna index and name
   cout << "Manually selected antenna: " << n << ": " << antenna_list[n] << endl;
```

The example sets the sample rate to 8 MHz with 8 times oversampling in RF. The resulting sample rates are then obtained using LMS_GetSampleRate(). This function obtains sample rate of the data stream to PC as well as the rate at which the RF signal is sampled in hardware. If only one of those rates is of interest, a 'NULL' can be safely passed in place of the other. In the example code bellow both rates are obtained and printed to the console.

```
   //Set sample rate to 8 MHz, preferred oversampling in RF 8x
   //This set sampling rate for all channels
   if (LMS_SetSampleRate(device, 8e6, 8) != 0)
      error();
   //print resulting sampling rates (interface to host , and ADC)
   float_type rate, rf_rate;
   if (LMS_GetSampleRate(device, LMS_CH_RX, 0, &rate, &rf_rate) != 0)  //NULL can be passed
      error();
    cout<<"\nHost interface sample rate: "<<rate/1e6<<" MHz\nRF ADC sample rate: "<<rf_rate/
1e6<<"MHz\n\n";
```

The next step that is performed is low-pass filter (LPF) configuration. At first, the example obtains a valid Rx LPF range via LMS_GetLPFBWRange() and prints it to the console. Then the LPF bandwidth for the first Rx channel is set to 8 MHz by calling LMS_SetLPFBW(). Note that the bandwidth passed to this function is bandwidth in RF.

```
//Example of getting allowed parameter value range
//There are also functions to get other parameter ranges (check LimeSuite.h)

//Get allowed LPF bandwidth range
lms_range_t range;
if (LMS_GetLPFBWRange(device,LMS_CH_RX,&range)!=0)
    error();
cout<<"RX LPF bandwitdh range: "<< range.min/1e6<<" - "<<range.max/1e6 << " MHz\n\n";

//Configure LPF, bandwidth 8 MHz
if (LMS_SetLPFBW(device, LMS_CH_RX, 0, 8e6) != 0)
    error();
```

The gain for the first Rx channel is set using normalized gain function LMS_SetNormalizedGain(). The gain range used by normalized gain functions is from 0.0 (minimum) to 1.0 (maximum). The gain can also be set in dB using LMS_SetGaindB(). Gain values are then read-back using GetNormalizedGain() to obtain normalized gain and GetGaindB() to obtain gain in dB. In functions that set/get gain in dB, '0' represents the minimum gain and the larger values should result in the RF signal being higher by approximately that value in dB.

```
//Set RX gain
if (LMS_SetNormalizedGain(device, LMS_CH_RX, 0, 0.7) != 0)
    error();
//Print RX gain
float_type gain; //normalized gain
if (LMS_GetNormalizedGain(device, LMS_CH_RX, 0, &gain) != 0)
    error();
cout << "Normalized RX Gain: " << gain << endl;

unsigned int gaindB; //gain in dB
if (LMS_GetGaindB(device, LMS_CH_RX, 0, &gaindB) != 0)
    error();
cout << "RX Gain: " << gaindB << " dB" << endl;
```

At the end of the configuration stage, automatic calibration of the first RX channel is performed via LMS_Calibrate(). In this example calibrations is performed for 8 MHz RF bandwidth. Test signal is also enabled in this example before streaming setup.

```
//Perform automatic calibration
if (LMS_Calibrate(device, LMS_CH_RX, 0, 8e6, 0) != 0)
    error();

//Enable test signal generation
//To receive data from RF, remove this line or change signal to LMS_TESTSIG_NONE
if (LMS_SetTestSignal(device, LMS_CH_RX, 0, LMS_TESTSIG_NCODIV8, 0, 0) != 0)
    error();
```

### 3.2.3 Sample streaming setup

Streaming setup in singleRX is very similar to the setup in basicRX. The main difference is that floating-point format is used for samples (dataFmt=lms_stream_t::LMS_FMT_F32). Note that buffer allocated for samples is also of float type. Floating-point format is not native for LimeSDR hardware and it is there only for convenience. The conversion is done in software.

```
//Streaming Setup

//Initialize stream
lms_stream_t streamId;
streamId.channel = 0; //channel number
streamId.fifoSize = 1024 * 1024; //fifo size in samples
streamId.throughputVsLatency = 1.0; //optimize for max throughput
streamId.isTx = false; //RX channel
streamId.dataFmt = lms_stream_t::LMS_FMT_F32; //32-bit floats
if (LMS_SetupStream(device, &streamId) != 0)
    error();

//Data buffers
const int bufersize = 10000; //complex samples per buffer
float buffer[bufersize * 2]; //must hold I+Q values of each sample
//Start streaming
LMS_StartStream(&streamId);
```

### 3.2.4 Receiving samples

Receiving samples is done in a while() loop. The loop in this example runs for 10 seconds and is very similar to the one in basicRX example. It reads samples from the stream FIFO via LMS_RecvStream() and plots them using GNUplot (if enabled). However, this example additionally prints some stream statistics every second. LMS_GetStreamStatus() is used to obtain information about the stream (link data, FIFO status, error counts). The example code bellow prints the link data rate and the percentage of FIFO filled to the console. Note that the error counters returned by LMS_GetStreamStatus() are reset each time this functions is called, so it returns the number of errors since the last call.

```
#ifdef USE_GNU_PLOT
    GNUPlotPipe gp;
    gp.write("set size square\n set xrange[-1:1]\n set yrange[-1:1]\n");
#endif
    auto t1 = chrono::high_resolution_clock::now();
    auto t2 = t1;

    while (chrono::high_resolution_clock::now() - t1 < chrono::seconds(10)) //run for 10 seconds
    {
        int samplesRead;
        //Receive samples
        samplesRead = LMS_RecvStream(&streamId, buffer, bufersize, NULL, 1000);
        //I and Q samples are interleaved in buffer: IQIQIQ...
            /*
                    INSERT CODE FOR PROCESSING RECEIVED SAMPLES
            */
        //Plot samples
#ifdef USE_GNU_PLOT
        gp.write("plot '-' with points\n");
        for (int j = 0; j < samplesRead; ++j)
            gp.writef("%f %f\n", buffer[2 * j], buffer[2 * j + 1]);
        gp.write("e\n");
```

```
        gp.flush();
#endif
        //Print stats (once per second)
        if (chrono::high_resolution_clock::now() - t2 > chrono::seconds(1))
        {
            t2 = chrono::high_resolution_clock::now();
            lms_stream_status_t status;
            //Get stream status
            LMS_GetStreamStatus(&streamId, &status);
            cout << "RX rate: " << status.linkRate / 1e6 << " MB/s\n"; //link data rate
            cout << "RX fifo: " << 100 * status.fifoFilledCount / status.fifoSize << "%" << endl;
//percentage of FIFO filled
        }
    }
```

At the end of the program, the device is closed the same way as in basicRX example.

```
//Stop streaming
LMS_StopStream(&streamId); //stream is stopped but can be started again with LMS_StartStream()
LMS_DestroyStream(device, &streamId); //stream is deallocated and can no longer be used

//Close device
LMS_Close(device);

return 0;
}
```

### 3.2.5 Application output

In setup stage application outputs parameter values that are obtained by API functions to console. Also, link data rate and percentage of stream FIFO filled is printed every second while streaming is running. Console output is shown in Figure 3. If GNUplot is enabled constellation of IQ samples is also plotted (Figure 4).

```
Devices found: 1
0: LimeSDR-USB, media=USB 3.0, module=STREAM, addr=1d50:6108, serial=0009060B00463119

[INFO] Estimated reference clock 30.7195 MHz
[INFO] Selected reference clock 30.720 MHz

Center frequency: 800 MHz
Available antennae:
0: NONE
1: LNA_H
2: LNA_L
3: LNA_W
Automatically selected antenna: 1: LNA_H
Manually selected antenna: 3: LNA_W

Host interface sample rate: 8 MHz
RF ADC sample rate: 64MHz

RX LPF bandwitdh range: 1.4 - 130 MHz

MCU algorithm time: 10 ms
MCU Ref. clock: 30.72 MHz
MCU algorithm time: 184 ms
Normalized RX Gain: 0.7
RX Gain: 49 dB
############################################################
Rx calibration using RSSI INTERNAL ON BOARD loopback
Rx ch.A @  800 MHz, BW: 8 MHz, RF input: LNAW, PGA: 10, LNA: 15, TIA: 3
Performed by: MCU
------------------------------------------------------------
MCU algorithm time: 0 ms
Current MCU firmware: 3, DC/IQ calibration full
MCU Ref. clock: 30.72 MHz
MCU algorithm time: 222 ms
[INFO] L
RX rate: 32.146 MB/s
RX fifo: 100%
RX rate: 32.146 MB/s
RX fifo: 100%
RX rate: 32.146 MB/s
RX fifo: 100%
RX rate: 32.146 MB/s
```

Figure 3: Console output of singleRX example

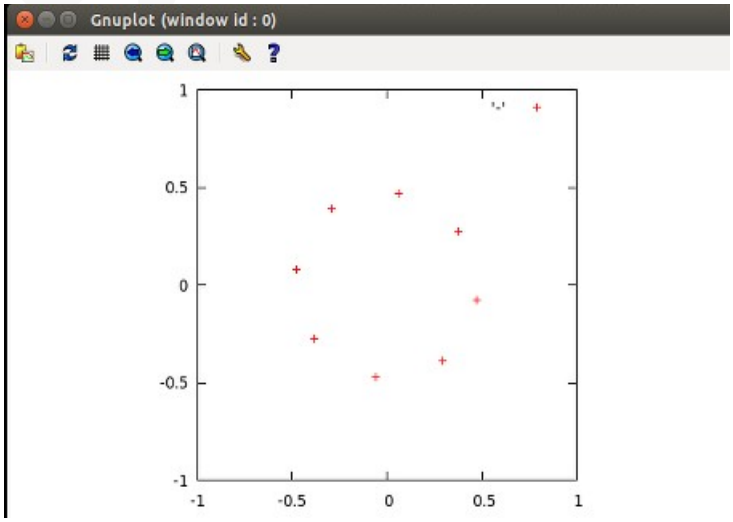


Figure 4: GNUplot output of singleRX example

## 3.3 Example 3: dualRXTX

Demonstrates receiving and sending of data using 2 RX and 2 TX channels. Compared to previous examples additionally demonstrates:

- Usage of multiple channels
- Transmitting data samples
- RX and TX synchronizations based on hardware timestamps

The example application connects to the first detected LimeSDR device, configures it and receives and sends samples for 10 seconds. The application retransmits received samples using synchronization based on timestamps to keep a constant offset between TX and RX at RF. The data transfer rate and FIFO status is printed every second while streaming is active.

### 3.3.1 Opening a device

The code at the beginning of dualRXTX example is the same as in basicRX example. The device is opened via LMS_Open() and working configuration is loaded by LMS_Init(). For more details refer to description of basicRX example.

```
#include "lime/LimeSuite.h"
#include <iostream>
#include <chrono>
#ifdef USE_GNU_PLOT
#include "gnuPlotPipe.h"
#endif

using namespace std;

//Device structure, should be initialize to NULL
lms_device_t* device = NULL;

int error()
{
    //print last error message
    cout << "ERROR:" << LMS_GetLastErrorMessage();
    if (device != NULL)
        LMS_Close(device);
    exit(-1);
}

int main(int argc, char** argv)
{
    //Find devices
    int n;
    lms_info_str_t list[8]; //should be large enough to hold all detected devices
    if ((n = LMS_GetDeviceList(list)) < 0) //NULL can be passed to only get number of devices
        error();

    cout << "Devices found: " << n << endl; //print number of devices
    if (n < 1)
        return -1;

    //open the first device
    if (LMS_Open(&device, list[0], NULL))
        error();
```

### 3.3.2 Device configuration

This example is meant to demonstrate usage of multiple channels, so at first it shows how to obtain the number of Rx/Tx channels. The code below obtains the number of channel via LMS_GetNumChannels() and prints it to the console.

```
//Get number of channels
if ((n = LMS_GetNumChannels(device, LMS_CH_RX)) < 0)
    error();
cout << "Number of RX channels: " << n << endl;
if ((n = LMS_GetNumChannels(device, LMS_CH_TX)) < 0)
    error();
cout << "Number of TX channels: " << n << endl;
```

After that goes the device configuration. The functions used in the configuration code bellow are described in previous examples and are not detailed in this section. The following steps are done:

- Modules required for 2 Rx and 2 Tx channels are enabled
- The Rx frequency for both channels is set to 1 GHz while Tx frequency is set to 1.2 GHz. Note that setting different frequencies for the first and the second channel is not supported as LMS7 uses single oscillator for both channels .
- Sample rate for all channels is set to 10 MHz.
- Gains are set for Rx and Tx channels
- Generation of test signals is enabled for RX channels.

```
//Enable RX channel
//Channels are numbered starting at 0
if (LMS_EnableChannel(device, LMS_CH_RX, 0, true) != 0)
    error();
if (LMS_EnableChannel(device, LMS_CH_RX, 1, true) != 0)
    error();
//Enable TX channels
if (LMS_EnableChannel(device, LMS_CH_TX, 0, true) != 0)
    error();
if (LMS_EnableChannel(device, LMS_CH_TX, 1, true) != 0)
    error();

//Set RX center frequency to 1 GHz
if (LMS_SetLOFrequency(device, LMS_CH_RX, 0, 1e9) != 0)
    error();
if (LMS_SetLOFrequency(device, LMS_CH_RX, 1, 1e9) != 0)
    error();
//Set TX center frequency to 1 GHz
//Automatically selects antenna port
if (LMS_SetLOFrequency(device, LMS_CH_TX, 0, 1.2e9) != 0)
    error();
if (LMS_SetLOFrequency(device, LMS_CH_TX, 1, 1.2e9) != 0)
    error();

//Set sample rate to 10 MHz, preferred oversampling in RF 4x
//This set sampling rate for all channels
if (LMS_SetSampleRate(device, 10e6, 4) != 0)
    error();

//Set RX gain
if (LMS_SetNormalizedGain(device, LMS_CH_RX, 0, 0.7) != 0)
    error();
if (LMS_SetNormalizedGain(device, LMS_CH_RX, 1, 0.7) != 0)
    error();
//Set TX gain
```

```
    if (LMS_SetNormalizedGain(device, LMS_CH_TX, 0, 0.4) != 0)
        error();
    if (LMS_SetNormalizedGain(device, LMS_CH_TX, 1, 0.4) != 0)
        error();

    //Enable test signals generation in RX channels
    //To receive data from RF, remove these lines or change signal to LMS_TESTSIG_NONE
    if (LMS_SetTestSignal(device, LMS_CH_RX, 0, LMS_TESTSIG_NCODIV4, 0, 0) != 0)
        error();
    if (LMS_SetTestSignal(device, LMS_CH_RX, 1, LMS_TESTSIG_NCODIV8F, 0, 0) != 0)
        error();
```

### 3.3.3 Sample streaming setup

In this example four streams are set-up in total (2 Rx and 2 Tx). Note that, all streams should be set-up before starting the streaming. The stream setup has already been explained in previous examples. The notable differences in this example are:

- Multiple Rx/Tx streams are configured. They have different channel parameter, while in previous examples only the first (0) channel was used.
- Previous examples were only dealing with Rx stream. Tx stream is set-up by setting 'isTx=true', while all other configuration is the same as for Rx stream.
- Parameter 'throughputVsLatency' is set to '0.5' instead of 1.0. This should provide good balance between throughput and latency.
- Also, this example is going to use a constant offset between Tx and Rx, so there are a couple of parameter to consider. If short offset between Rx and Tx is required, the latency needs to be minimized. On the other hand if delay between Rx and Tx is long, sufficiently large FIFO buffers are required as samples will be staying in them waiting to be sent.

```
//Streaming Setup

    const int chCount = 2; //number of RX/TX streams
    lms_stream_t rx_streams[chCount];
    lms_stream_t tx_streams[chCount];
    //Initialize streams
    //All streams setups should be done before starting streams. New streams cannot be set-up if at
least stream is running.
    for (int i = 0; i < chCount; ++i)
    {
        rx_streams[i].channel = i; //channel number
        rx_streams[i].fifoSize = 1024 * 1024; //fifo size in samples
        rx_streams[i].throughputVsLatency = 0.5; //something in the middle
        rx_streams[i].isTx = false; //RX channel
        rx_streams[i].dataFmt = lms_stream_t::LMS_FMT_I12; //12-bit integers
        if (LMS_SetupStream(device, &rx_streams[i]) != 0)
            error();
        tx_streams[i].channel = i; //channel number
        tx_streams[i].fifoSize = 1024 * 1024; //fifo size in samples
        tx_streams[i].throughputVsLatency = 0.5; //something in the middle
        tx_streams[i].isTx = true; //TX channel
        tx_streams[i].dataFmt = lms_stream_t::LMS_FMT_I12; //12-bit integers
        if (LMS_SetupStream(device, &tx_streams[i]) != 0)
            error();
    }

    //Initialize data buffers
    const int bufersize = 1024 * 8; //complex samples per buffer
    int16_t * buffers[chCount];
    for (int i = 0; i < chCount; ++i)
```

```
    {
        buffers[i] = new int16_t[bufersize * 2]; //buffer to hold complex values (2*samples))
    }

    //Start streaming
    for (int i = 0; i < chCount; ++i)
    {
        LMS_StartStream(&rx_streams[i]);
        LMS_StartStream(&tx_streams[i]);
    }
```

### 3.3.4 Streaming samples

The streaming code in this section has a lot of similarities to the code in singleRX example. Once again only the things that were not covered by previous examples are explained in this section.

First of all, there are metadata structures for Tx and Rx. They are going to be passed to LMS_RecvStream() and LMS_SendStream() functions. In Rx metadata structure is going to be used to obtain hardware timestamp of the samples received. In Tx metadata structure will be used to signal that samples should be sent at s specific hardware timestamp. To do that 'waitForTimestamp' parameter is set to 'true' in Tx metadata structure.

```
    lms_stream_meta_t rx_metadata;//Use metadata for additional control over sample receive
function behavior
    rx_metadata.flushPartialPacket = false; //currently has no effect in RX
    rx_metadata.waitForTimestamp = false; //currently has no effect in RX

    lms_stream_meta_t tx_metadata; //Use metadata for additional control over sample send function
behavior
    tx_metadata.flushPartialPacket = false; //do not force sending of incomplete packet
    tx_metadata.waitForTimestamp = true; //Enable synchronization to HW timestamp

#ifdef USE_GNU_PLOT
    GNUPlotPipe gp;
    gp.write("set size square\n set xrange[-2050:2050]\n set yrange[-2050:2050]\n");
#endif
    auto t1 = chrono::high_resolution_clock::now();
    auto t2 = t1;
```

The streaming code in while() loop receives samples from two Rx channels and retransmits them with constant offset to two Tx channels. The following steps are performed for each channel:

- Samples are read to buffer from the Rx FIFO via LMS_RecvStream(). The number of samples read is obtained from the function return value and the metadata structure should contain the hardware timestamp of the first sample in the buffer.
- A constant offset is added to Rx timestamp and the resulting value is written to Tx metadata structure. Note that if the packet of samples arrives to hardware to late they are not transmitted to RF at all and 'L' character is printed to the console.
- The received buffer, the number of samples read, and Tx metadata (containing timestamp at which the samples should be transmitted to RF) are then passed to LMS_SendStream() function.

```
    while (chrono::high_resolution_clock::now() - t1 < chrono::seconds(10)) //run for 10 seconds
    {
        for (int i = 0; i < chCount; ++i)
        {
            int samplesRead;
            //Receive samples
            samplesRead=LMS_RecvStream(&rx_streams[i], buffers[i], bufersize,&rx_metadata, 1000);
            //Send samples with 1024*256 sample delay from RX (waitForTimestamp is enabled)
            tx_metadata.timestamp = rx_metadata.timestamp + 1024 * 256;
            LMS_SendStream(&tx_streams[i], buffers[i], samplesRead, &tx_metadata, 1000);
        }
```

The remaining code in the while() loop plots samples using GNUplot (if enabled) as well as output these parameters:

- RX data link rate
- Filled percentage of the first channel RX FIFO
- TX data link rate
- Filled percentage of the first channel TX FIFO

```
        //Print stats every 1s
        if (chrono::high_resolution_clock::now() - t2 > chrono::seconds(1))
        {
#ifdef USE_GNU_PLOT
            //Plot samples
            t2 = chrono::high_resolution_clock::now();
            gp.write("plot '-' with points");
            for (int i = 1; i < chCount; ++i)
                gp.write(", '-' with points\n");
            for (int i = 0; i < chCount; ++i)
            {
                for (uint32_t j = 0; j < bufersize / 8; ++j)
                    gp.writef("%i %i\n", buffers[i][2 * j], buffers[i][2 * j + 1]);
                gp.write("e\n");
                gp.flush();
            }
#endif
            //Print stats
            lms_stream_status_t status;
            LMS_GetStreamStatus(rx_streams, &status); //Obtain RX stream stats
            cout << "RX rate: " << status.linkRate / 1e6 << " MB/s\n"; //link data rate (both channels))
            cout << "RX 0 FIFO: " << 100 * status.fifoFilledCount / status.fifoSize << "%" << endl;
//percentage of RX 0 fifo filled

            LMS_GetStreamStatus(tx_streams, &status); //Obtain TX stream stats
            cout << "TX rate: " << status.linkRate / 1e6 << " MB/s\n"; //link data rate (both channels))
            cout << "TX 0 FIFO: " << 100 * status.fifoFilledCount / status.fifoSize << "%" << endl;
//percentage of TX 0 fifo filled
        }
    }
```

Finally, after the loop all streams are stopped and device is closed.

```
    //Stop streaming
    for (int i = 0; i < chCount; ++i)
    {
        LMS_StopStream(&rx_streams[i]); //stream is stopped but can be started again with
LMS_StartStream()
        LMS_StopStream(&tx_streams[i]);
    }
    for (int i = 0; i < chCount; ++i)
    {
```

```
        LMS_DestroyStream(device, &rx_streams[i]); //stream is deallocated and can no longer be
used
        LMS_DestroyStream(device, &tx_streams[i]);
        delete[] buffers[i];
    }

    //Close device
    LMS_Close(device);

    return 0;
}
```

### 3.3.5 Application output

The code at the beginning of dualRXTX example is the same as in basicRX example. The device is opened via LMS_Open() and working configuration is loaded by LMS_Init(). For more details refer to description of basicRX example.

```
Devices found: 1
[INFO] Estimated reference clock 30.7196 MHz
[INFO] Selected reference clock 30.720 MHz
Number of RX channels: 2
Number of TX channels: 2
[INFO] L
RX rate: 60.2604 MB/s
RX 0 FIFO: 2%
TX rate: 58.753 MB/s
TX 0 FIFO: 13%
RX rate: 60.2604 MB/s
RX 0 FIFO: 1%
TX rate: 60.2604 MB/s
TX 0 FIFO: 13%
RX rate: 60.2604 MB/s
RX 0 FIFO: 1%
TX rate: 60.2604 MB/s
TX 0 FIFO: 13%
```

Figure 5: Console output of dualRXTX example

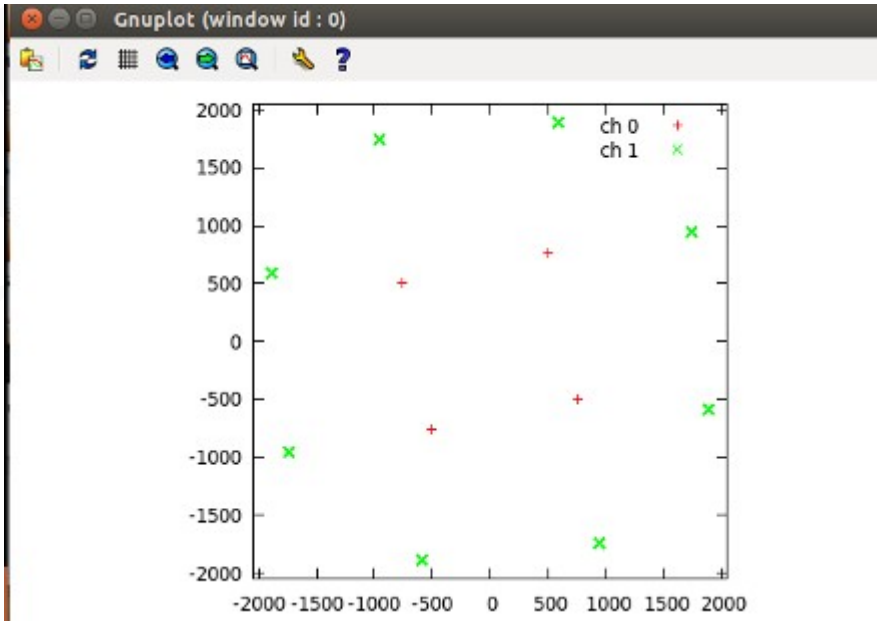


Figure 6: GNU plot output of dualRXTX example