Descripción General

En un sistema operativo linux se ha instalado la libreria de LimeSuite la cual es un propuesta de código abierto para poder desarrollar sobre un software define Radio conocido como Lime7002M.
https://github.com/myriadrf/LimeSuite

El projecto cuenta con las librerias necesarias para trabajar en distintos modos; entre ellos esta : GUI, sistema embebido en C y C++.

Funcionamiento de codigo C++

Una vez instalada la libreria con Cmake install, se puede usar la misma libreria llamandola como libreria reconocida por el sistema operativo.

Preparación libreria con CMAKE

```
cmake_minimum_required(VERSION 3.1.0)

set(CMAKE_CXX_STANDARD 11)

add_executable(basicTX basicTX.cpp)
set_target_properties(basicTX PROPERTIES
RUNTIME_OUTPUT_DIRECTORY
"${CMAKE_BINARY_DIR}/../executables")
target_link_libraries(basicTX LimeSuite)
```

Posteriormente se mostrara el codigo basicTx.cpp donde se incluiran sus directorios de librearias

```
/**
    @file   basicTX.cpp
    @brief  minimal TX example
 */
#include <iostream>
#include <chrono>
#include <math.h>
#include "lime/LimeSuite.h"
```

Funciones de cabecera y variables globales

La estrucutra **lms_device_t** es la que nos dara acceso a todos los recursos del dispositivo, tanto a su conexión usb, como sus funciones para configurar propiedades de la antena.

Se agrega una funcion de error en caso de que haya problemas con el hilo de comunicación y no se quede perpetuamente abierto produciendo un death lock.

```
using namespace std;

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

int error()
{
    if (device != NULL)
    LMS_Close(device);
    exit(-1);
}
```

Configuración de variables de radio

Una vez entrando al main se elige la configuración del radio con la que se va operar, en este codigo se dejan constantes los valores ya que se pretende dejarlos sin cambios, pero pueden ser perfectamente mutables sin ningun problema y configurables en tiempo de ejecución.

```
int main(int argc, char** argv)
{
    const double frequency = 500e6;
  //center frequency to 500 MHz
    const double sample_rate = 4e6;
  //sample rate to 5 MHz
    const double tone_freq = 1e6;
    //tone frequency
    const double f_ratio = tone_freq/sample_rate;
```

Inicializar la conexión al radio

Se obtiene la lista de posibles radios, asi tambien se usa la función device list para encontrar el ID del radio que el kernel provee, una vez encontrado el ID adecuado de usb se abre la conexión y se busca inicializar el dispostivo, cabe recalcar que la inicialización no genera calibración, su unica función es alocar memoria necesaria para trabajar con los paquetes de USB e internamente de SPI.

```
//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
if (n < 1)
    return -1;

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

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

Configurar radio con parametros

En esta sección se manda a llamar las funcion que internetamente tienen el arreglo de registros internos que se le llamaran al micro, estas API permiten directamente y sin tanto complicación , realizar configuración usualues del dispotivo. Las funciones tienen un diseño de este estilo.

LMS_<Configuración>(device, <TX/RX>, Channel, <valor numerico>)

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

//Set sample rate
if (LMS_SetSampleRate(device, sample_rate, 0)!=0)
    error();
cout << "Sample rate: " << sample_rate/1e6 << " MHz" << endl;

//Set center frequency
if (LMS_SetLOFrequency(device,LMS_CH_TX, 0, frequency)!=0)
    error();
cout << "Center frequency: " << frequency/1e6 << " MHz" << endl;

//select TX1_1 antenna
if (LMS_SetAntenna(device, LMS_CH_TX, 0, LMS_PATH_TX1)!=0)
    error();

//set TX gain
if (LMS_SetNormalizedGain(device, LMS_CH_TX, 0, 0.7) != 0)
    error();

//calibrate Tx, continue on failure
LMS_Calibrate(device, LMS_CH_TX, 0, sample_rate, 0);
```

Generación de señal por software

En la primera parte del codigo se hace una configuración para poner el tamaño del buffer que se va usar para la transmición, asi como el tipo de dato transmitido (float). Despues se genera la señal de tono , que sera transmitida, la frecuencia de la señal esta relaciónada con f_ratio que se encuentra en la sección de “*Configuración de variables de radio”*

```
//Streaming Setup

   lms_stream_t tx_stream;                 //stream structure
   tx_stream.channel = 0;                  //channel number
   tx_stream.fifoSize = 256*1024;
          //fifo size in samples
   tx_stream.throughputVsLatency = 0.5;
    //0 min latency, 1 max throughput
   tx_stream.dataFmt = lms_stream_t::LMS_FMT_F32;
 //floating point samples
   tx_stream.isTx = true;                  //TX channel
   LMS_SetupStream(device, &tx_stream);

   //Initialize data buffers
   const int buffer_size = 1024*8;
   float tx_buffer[2*buffer_size];
     //buffer to hold complex values (2*samples))
   for (int i = 0; i <buffer_size; i++) {
      //generate TX tone
       const double pi = acos(-1);
       double w = 2*pi*i*f_ratio;
       tx_buffer[2*i] = cos(w);
       tx_buffer[2*i+1] = sin(w);
   }
```

Streaming

Se hace una carga de la señal al buffer, y el streaming se deja por 10 segundos

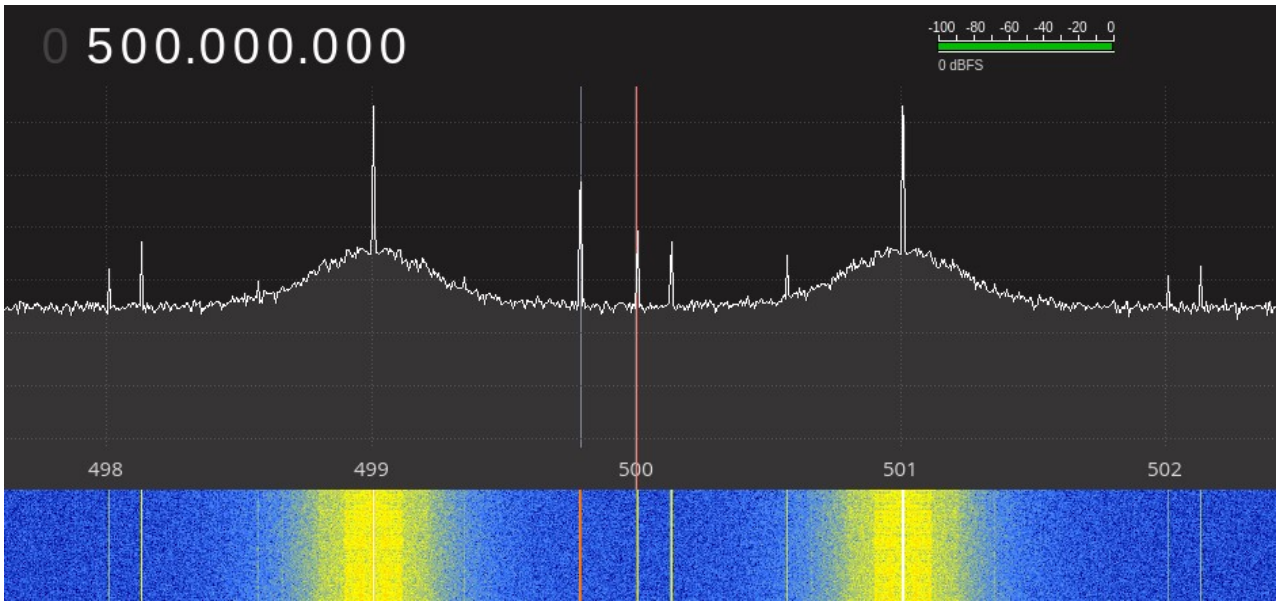
```
LMS_StartStream(&tx_stream);         //Start streaming
//Streaming
auto t1 = chrono::high_resolution_clock::now();
auto t2 = t1;
while (chrono::high_resolution_clock::now() - t1 < chrono::seconds(10))
 //run for 10 seconds
{
    //Transmit samples
    int ret = LMS_SendStream(&tx_stream, tx_buffer, send_cnt, nullptr, 1000);
    if (ret != send_cnt)
        cout << "error: samples sent: " << ret << "/" << send_cnt << endl;
    //Print data rate (once per second)
    if (chrono::high_resolution_clock::now() - t2 > chrono::seconds(1))
    {
        t2 = chrono::high_resolution_clock::now();
        lms_stream_status_t status;
        LMS_GetStreamStatus(&tx_stream, &status);  //Get stream status
        cout << "TX data rate: " << status.linkRate / 1e6 << " MB/s\n"; //link data rate
    }
}
```

Una vez terminada la carga y el envio por paquete se dice el rate de transmición en MB/s

```
sample count per send call: 8192
TX data rate: 8.06093 MB/s
TX data rate: 8.03635 MB/s
TX data rate: 8.03635 MB/s
TX data rate: 8.03635 MB/s
TX data rate: 8.03635 MB/s
TX data rate: 8.03635 MB/s
TX data rate: 8.03635 MB/s
TX data rate: 8.03635 MB/s
TX data rate: 8.03635 MB/s
```

Resultados

Para poder visualizar los resultados, se utilizo una HackRF con el software gqrx. En la imagen inferior se puede mostrar el resultado de la señal, la cual esta espaciada por un 1Mhz del centro de la frequencia principal, dado la frequencia de tono de la señal.



Cierre

Una ves terminado el tiempo para streamear, se para el proceso y se libera el buffer reservado, despues se deshabilitan los canales de transmición y se cierra la comunicación por USB.

```
//Stop streaming
LMS_StopStream(&tx_stream);
LMS_DestroyStream(device, &tx_stream);

//Disable TX channel
if (LMS_EnableChannel(device, LMS_CH_TX, 0, false)!=0)
    error();

//Close device
if (LMS_Close(device)==0)
    cout << "Closed" << endl;
return 0;
```