LMS7002M Python Package
pyLMS7002M

Introduction

Python package pyLMS7002M is platform-independent, and is intended for fast prototyping and algorithm development. It provides low level register access and high level convenience functions for controlling the LMS7002M chip and evaluation boards. Supported evaluation boards are:

- LMS7002_EVB
- LimeSDR

The package consists of Python classes which correspond to physical or logical entities. For example, each module of LMS7002M (AFE, SXT, TRF, ...) is a class. The LMS7002M chip is also a class containing instances of on-chip modules. The evaluation board class contains instances of on-board chips, such as LMS7002, ADF4002, etc. Classes follow the hierarchy and logical organization from evaluation board down to on-chip register level.

Installation

The pyLMS7002M package is installed in a usual way:

python setup.py install

Module installation can be verified from Python:

python

>>> from pyLMS7002M import *

If there is no error, the module is correctly installed.

USB Communication

USB communication can be established in two ways:

- **Option 1 – Direct USB communication** (recommended)

  Python communicates directly to the USB driver. This is the simplest and recommended option.

  Pros: Lightweight solution, simple installation

  Cons:

  Windows users: Some other applications that are communicating with LimeSDR (such as LimeSuiteGUI and PothosSDR) use LimeSDR-USB Windows driver (https://github.com/myriadrf/Windows-drivers). The pyLMS7002M cannot use this driver. This means that the user has to change the driver each time he/she switches from using pyLMS7002M library to using e.g. LimeSuiteGUI, and vice versa.

  Linux users: There are no problems, since all applications use the same driver.

- **Option 2 – USB communication using LimeAPI**

  Python communicates to the LimeAPI library, which communicates to the USB driver.

  Pros:

  Windows users: No need to change the drivers. LimeSDR-USB Windows drivers (https://github.com/myriadrf/Windows-drivers) can be used both for pyPLS7002M library, and other applications such as LimeSuiteGUI and PothosSDR.

  Linux users: There is no advantage, since all applications use the same driver.

  Cons: More complicated installation and setup.

Note: Only one method of USB communication should be used, since they are not compatible on Windows systems.
In the following the installation and usage will be described for both options.

Option 1 – Direct USB Communication (Recommended)

This is the recommended option. In case you opt for direct USB communication the following section can be skipped.

Linux users: Most Linux distributions already have libusb installed.

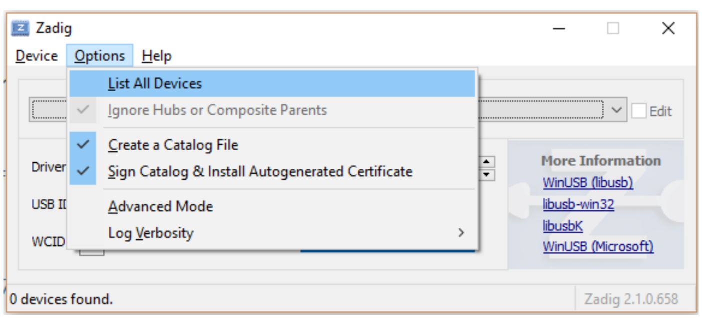
Windows users: The default Windows drivers for LimeSDR boards are not compatible with the Python module pyUSB. Drivers can be changed by using software Zadig. Zadig for Windows Vista/Win7/Win8 32/64 bit can be downloaded from:

http://zadig.akeo.ie/downloads/zadig_2.1.0.exe

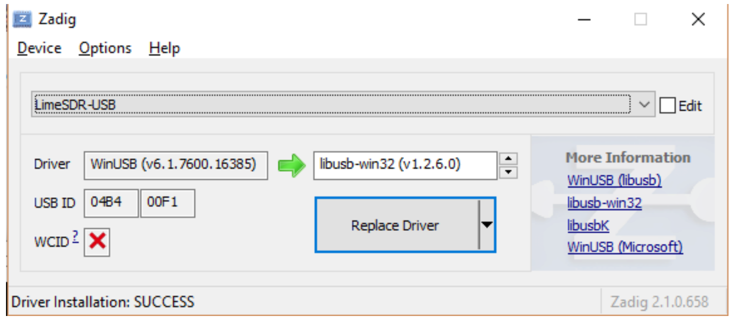
Zadig for Windows XP can be downloaded from:

http://zadig.akeo.ie/downloads/zadig_xp_2.1.0.exe

When Zadig is run, click on Options->List All Devices, as shown in the figure below.



Then select the LimeSDR and libusb-win32 from drop-down lists and click on Replace Driver button.



If the user wants to use other applications, such as LimeSuiteGUI, or PothosSDR, then the LimeSDR-USB Windows driver has to be restored. Please refer to the driver installation instructions for these applications (http://wiki.myriadrf.org/Lime_Suite#USB_driver).

Option 2 – USB Communication Using LimeAPI

This option is recommended only for Windows users that frequently use both pyLMS7002M python library and other applications for communication with LimeSDR, such as LimeSuiteGUI, or PothosSDR. By installing this option the same LimeSDR-USB Windows driver can be used in both cases.

For other users this is not the recommended option, and this section should be skipped.

Both Windows and Linux users that, for any reason, want to use communication via LimeAPI library can do it as described in the following.

Communication with the LimeAPI is enabled through the cyLimeLib library.

It is assumed that LimeSuite shared library is already installed.

Cython library is required, and can be install by executing:

pip install Cython

Windows requirements:

For Windows users another requirement is to install "Microsoft Visual C++ Compiler for Python 2.7" that can be downloaded from http://aka.ms/vcpython27.

In Windows the library relies on the LimeSuite.dll and the FTD3XX.dll libraries. It is recommended to download and install PothosSDR from http://downloads.myriadrf.org/builds/PothosSDR/?C=M;O=D which will provide up-to-date required libraries. Other option is to copy these files from the cyLimeLib folder to a location that is in the system path.

Linux requirements:

Linux users should have libLimeSuite.so installed. The easiest way to get the current version of libLimeSuite.so is to build and install the latest LimeSuiteGUI, which can be obtained from https://github.com/myriadrf/LimeSuite. Archive libLimeSuite.tar.gz with precompiled library for 64 bit Ubuntu is provided in cyLimeLib/linux directory. It should be extracted in a directory which is in linker search path, e.g. /usr/lib or a directory included in $LD_LIBRARY_PATH.

To use the library, the provided module should be built and installed by executing (from cyLimeLib directory):

```
python setup.py install
```

Module installation can be verified from Python:

```
python
>>> from cyLimeLib import *
```

If there is no error, the module is correctly installed.

Basic usage

The first step is to connect to the evaluation board:

```
>>> from pyLMS7002M import *
```

List of COM ports with LMS7002_EVB attached can be obtained as:

```
>>> ports = LMS7002_EVB.findLMS7002()
>>> lms7002_evb = LMS7002_EVB(portName=ports[0])
```

Connection to LimeSDR can be established with:

```
>>> limeSDR = LimeSDR()
```

Now that the board is connected, the on-board chips can be used. For example, board clock can be synchronized to external 10 MHz reference by configuring the on-board ADF4002.

```
>>> adf4002 = limeSDR.ADF4002
>>> adf4002.enable() # Configure and enable the on-board ADF4002
```

The ADF4002 can be disabled with:

```
>>> adf4002.disable() # Disable the on-board ADF4002
```

On-board LMS7002M chip can be accessed as:

```
>>> lms7002 = limeSDR.LMS7002
```

Registers can be accessed with overloaded [ ] operator:

```
>>> lms7002[0x2f]
Register : ChipVer 0x002F
VER<4:0>        00111           (0x0007 << 11)      (7 << 11)
REV<4:0>        00001           (0x0001 << 6) (1 << 6)
MASK<5:0>       000000          (0x0000 << 0) (0 << 0)
Register value  0011100001000000        (0x3840)
```

Registers can be accessed by address as shown in the previous example, or by name:

```
>>> lms7002['ChipVer']
```

Register definition can be accessed with the help function:

```
>>> lms7002['ChipVer'].help()
REGISTER    ChipVer    0x002F
   BITFIELD   VER<4:0>
      POSITION=<15:11>
      VALUE=00111
      MODE=R
      #!  Chip version. Read only.
      #!  00111 - Chip version is 7
   ENDBITFIELD
   BITFIELD   REV<4:0>
      POSITION=<10:6>
      VALUE=00001
      MODE=R
      #!  Chip revision. Read only.
      #!  00001 - Chip revision is 1
   ENDBITFIELD
   BITFIELD   MASK<5:0>
      POSITION=<5:0>
      VALUE=000000
      MODE=R
      #!  Chip mask. Read only.
      #!  000000 - Chip mask is 0
   ENDBITFIELD
ENDREGISTER
```

Individual bit-fields can be accessed also:

```
>>> chipVer=lms7002['ChipVer']
>>> chipVer['REV<4:0>']
   1
```

Register value can be written directly:

```
>>> lms7002['TRF_CFG']=0x3409
```

Single bitfield can also be changed:

```
>>> lms7002['TRF_CFG']['EN_G_TRF']=1
```

Read/write operations to LMS7002M SPI are controlled by MAC flag, so the result of previous operation depends on the value of MAC, which can be accessed as:

```
>>> lms7002.MAC
         1
```

Although it is supported, setting the MAC value and writing to registers as shown in the previous examples is not encouraged. A better way is to use the benefits of object-oriented approach. Each module in LMS7002M has an instance for each channel. For example, TRF channel A can be accessed:

```
>>> TRF_A = lms7002.TRF['A']
```

EN_G_TRF bitfield can now be written with:

```
>>> TRF_A.EN_G_TRF = 1
```

Now the MAC value will be automatically set to the correct value to access channel A. Some bitfields have values which can be interpreted as a state or action. For example, TxTSP has bitfields which control whether the block (GFIR, CMIX, ...) is bypassed or used. Besides the numeric value, where applicable, assigning meaningful strings are also supported to improve code readability. In the TxTSP example, complex mixer can be bypassed with

```
>>> TxTSP_A = lms7002.TxTSP['A']
>>> TxTSP_A.CMIX_BYP = 1
```

To improve the code readability, the value can be specified as a string

```
>>> TxTSP_A.CMIX_BYP = 'BYP'
```

Complex mixer can be configured to be used with

```
>>> TxTSP_A.CMIX_BYP = 'USE'
```

Since each bitfield checks whether the given value is valid, the list or range of valid values can be obtained by examining the source code, or by triggering an error:

```
>>> TxTSP_A.CMIX_BYP = '?'
   ValueError: Value must be [0,1,'USE', 'BYP']
```

Some bitfields use two's complement or sign-magnitude data format. For example, bitfield DCCORRI<7:0> uses two's complement data format. Convenience functions have been written to facilitate automatic data format conversion, so the value of DC correction can be set with:

```
>>> TxTSP_A.DCCORRI = -19
```

and the Python code will convert the given value to two's complement format, pack it into register and write the register value to the LMS7002M chip. Similarly, there are configuration values which are split into two registers, such as FRAC_SDM_L<15:0> and FRAC_SDM_H<3:0>. Convenience functions allow the user to write the intended value and the Python code will perform the neccessary steps to convert the given value to the format expected by the chip. For example, fractional part of SDM can be set as:

```
>>> lms7002.SX['T'].FRAC_SDM = 93489
```

The Python code will split the given value into 4 MSB bits and 16 LSB bits and write them to appropriate registers. Reading the FRAC_SDM will do the opposite – read the registers and convert the value to integer as MSB<<16+LSB.

High level functions

Besides the basic functionality for reading/writing registers, high level functions are also provided to simplify the chip configuration. For example, configuring and locking the PLL to a given frequency requires a sequence of steps. The pyLMS7002M package provides the high level functions for the following operations:

- Configuration and locking of SXT/SXR/CGEN
- Chip configuration from ini files generated by LMS7002 GUI
- Programming the 8051 MCU

Clock generator can be configured and locked to a given frequency, in this case 300 MHz, with a single command:

```
>>> lms7002.CGEN.setCLK(300e6)
```

Transmit and receive PLLs can be configured in a similar manner:

```
>>> lms7002.SX['T'].setFREQ(1.2e9)
>>> lms7002.SX['R'].setFREQ(2.4e9)
```

Chip configuration can be read from ini file and programmed into LMS7002M with:

```
>>> lms7002.readIniFile('chipConf.ini', writeToChip=True)
```

On-chip 8051 MCU SRAM can be programmed with a given hex file with:

```
>>> lms7002.mSPI.loadHex('hexFile.hex', mode='SRAM')
```