
Lime Microsystems Limited

Surrey Tech Centre
Occam Road
The Surrey Research Park
Guildford, Surrey GU2 7YG
United Kingdom



Tel: +44 (0) 1483 685 063
Fax: +44 (0) 1428 656 662
e-mail: enquiries@limemicro.com

LMS7 API

Chip version:	LMS7002Mr2
Chip revision:	01
Document version:	02
Document revision:	01
Last modified:	03/07/2015 19:00:00

Contents

1. Communication with board.....	4
2. Configuration file.....	5
2.1 Description	5
2.2 Properties and values.....	5
2.3 Sections	5
2.4 Duplicate values	5
3. Registers writing and reading	6
4. PLL & CGEN configuration	7
4.1 PLL.....	7
4.2 CGEN.....	7
4.3 Tuning	7
5. RxTSP & TxTSP configuration	9
5.1 NCO frequencies	9
5.2 NCO phase offsets.....	9
5.3 GFIR coefficients	9
6. Transmitter & receiver calibration	11
6.1 Transmitter	11
6.2 Receiver.....	11
7. Filter tuning	12
7.1 Transmitter	12
7.2 Receiver.....	12
8. Additional functionality	13
8.1 Interface frequency.....	13

Revision History

Version 01r00

Released: 3 July, 2015

Initial version.

1

Communication with board

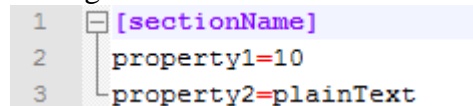
All library communications with the chip are done by using `LMScomms` class. It has a packet transfer function which takes and returns generic arrays of bytes and transfers them to chip according to used communications protocol. This class can be extended to support other needed communication types or protocols. Generic packet specifies which communications command is to be performed and how many bytes should be sent from it's output buffer. Upon transfer completion the packet's input buffer should be updated with received data and number of bytes received. The packet's status should be changed to represent overall transfer status, indicating success or any other failure. Currently `LMScomms` support virtual COM and USB connections using `LMS64C` protocol.

2

Configuration file

2.1 Description

This chapter specifies INI configuration files formatting and structure. The INI file is simple text file with basic structure composed of sections, properties and values. It is human-readable and simple to parse as shown in Figure 1.



```
1 [sectionName]
2 property1=10
3 property2=plainText
```

Figure 1 INI file example

2.2 Properties and values

The basic element contained in INI file is the property and value pair. Property and value pair must be in the same line and is delimited with equals sign (=). The property name is on the left side, and the value is on the right side. Each pair must be declared in a new line.

2.3 Sections

Properties can be grouped into arbitrarily named sections. The section start is declared in a line as name surrounded by square brackets [and]. All properties after the section declaration are associated with it. There is no “end of section” delimiter, section ends at the next section declaration, or at the end of the file. Sections may not be nested.

2.4 Duplicate values

The same property name can be used in multiple sections. Within a section if duplicate property names are encountered the value is overwritten with the last declared value.

3

Registers writing and reading

Library keeps track of register operations and has local copy of the chip's registers map. When chip is not connected all operations are done only with local registers copy. When chip is connected its internal registers might differ from the local copy, so it is advised to synchronize them using **UploadAll()** to change chip registers or **DownloadAll()** to update local copy. Registers manipulations are performed by using the following functions, **these functions are not thread safe**:

SPI_write(uint16_t address, uint16_t data):

Writes given data to selected register address.

SPI_read(uint16_t address, liblms7_status *status = 0):

Reads whole register value from selected address, the status is optional, it can be used to get operation status.

Get_SPI_Reg_bits(uint16_t address, uint8_t msb, uint8_t lsb):

Uses **SPI_read** function to read selected register's value and returns its desired bits interval masked and shifted to right by **lsb** bits.

Modify_SPI_Reg_bits(uint16_t address, uint8_t msb, uint8_t lsb, uint16_t value):

Uses **SPI_read** and **SPI_write** functions to modify selected register's bits in given interval. The new given value is shifted left by **lsb** bits and replaces given interval bits in value received from chip.

Get_SPI_Reg_bits(const LMS7Parameter ¶m):

Alternative function to get chip parameter value by using predefined parameter description structure.

Modify_SPI_Reg_bits(const LMS7Parameter ¶m, const uint16_t value):

Alternative function to set chip parameter value by using predefined parameter description structure.

4

PLL & CGEN configuration

4.1 PLL

PLL frequency is configured by using **SetFrequencySX**(bool tx, float_type freq_MHz, float_type refClk_MHz) function. It requires to supply transmitter or receiver selection, desired frequency in MHz and a reference frequency in MHz to be used for coefficients calculations. This function also performs VCO tuning procedure. This function automatically changes chip's MAC parameter according to transmitter or receiver selection, and restores it upon configuration completion. If desired frequency is not achievable, function will return **LIBLMS7_FREQUENCY_OUT_OF_RANGE** meaning the frequency is outside of available range, or **LIBLMS7_CANNOT_DELIVER_FREQUENCY** meaning the desired frequency is in available range, but VCO cannot provide it.

Reading PLL frequency can be done by using **GetFrequencySX_MHz**(bool Tx, float_type refClk_MHz) function. It requires to supply transmitter or receiver selection and currently used PLL reference frequency in MHz, the function returns PLL actual frequency in MHz.

4.2 CGEN

CGEN frequency is configured by using **SetFrequencyCGEN**(const float_type freq_MHz) function. It requires only desired frequency in MHz, the reference clock used for coefficients calculation is taken from the last Rx PLL frequency calculations call. If Rx PLL frequency was not previously set, the reference clock defaults to 30.72 MHz. This function also performs VCO tuning procedure.

Reading CGEN frequency can be done by using **GetFrequencyCGEN_MHz**() function. It returns current CGEN frequency in MHz, the frequency is calculated by using reference clock from last Rx PLL frequency setting.

4.3 Tuning

VCOs are automatically tuned after setting their frequency. Tuning process can be repeated by calling **TuneVCO(VCO_Module module)** and selecting which VCO should be tuned, the available selections are VCO_CGEN, VCO_SXR, VCO_SXT. The chip's MAC control is automatically changed depending on VCO selection, and restored after procedure.

5

RxTSP & TxTSP configuration

5.1 NCO frequencies

TSP NCO can be configured by using **SetNCOFrequency(bool tx, uint8_t index, float_type freq_MHz)** function. It requires transmitter or receiver selection, desired NCO index from 0 to 15 and the desired frequency in MHz. The reference clock used for NCO coefficients is calculated depending on last used Rx PLL frequency and currently active clocks switching. When NCO memory table MODE is 0 (FCW table), the phase offset can be set by using **SetNCOPhaseOffsetForMode0(bool tx, float_type angle_Rad)** function.

NCO frequency can be read by using **GetNCOFrequency_MHz(bool tx, uint8_t index)** function. It requires transmitter or receiver selection and desired NCO index from 0 to 15, the returned frequency is given in MHz.

5.2 NCO phase offsets

TSP NCO phase offsets can be configured by using **SetNCOPhaseOffset(bool tx, uint8_t index, float_type angle_Rad)** function. It requires transmitter or receiver selection, desired NCO index from 0 to 15 and the phase offset in radians.

NCO phase offsets can be read by using **GetNCOPhaseOffset_Rad(bool tx, uint8_t index)** function. It requires transmitter or receiver selection and NCO index from 0 to 15, the returned phase offset is given in radians.

5.3 GFIR coefficients

GFIR coefficients can be set by using **SetGFIRCoefficients(bool tx, uint8_t GFIR_index, const int16_t *coef, uint8_t coefCount)** function. It requires transmitter or receiver selection, GFIR index from 0 to 2 (0-GFIR1, 1-GFIR2, 2-GFIR3), array of signed integer coefficients and number of coefficients to upload. The maximum number of

coefficients is different for each GFIR. **Setting coefficients does not change GFIR*_L or GFIR*_N parameters, they have to be set manually.**

GFIR coefficients can be read by using **GetGFIRCoefficients(bool tx, uint8_t GFIR_index, int16_t *coef, uint8_t coefCount)** function. It requires transmitter or receiver selection, GFIR index from 0 to 2, destination array for coefficients and number of coefficients to read.

6

Transmitter & receiver calibration

6.1 Transmitter

Calibration requires user to configure transmitter to working state and choose desired bandwidth in MHz. The calibration can be performed by using **CalibrateTx(float_type bandwidth_MHz)**, it automatically changes **MAC** parameter and restores it after procedure ends. If the calibration succeeds **DCCORRI**, **DCCORRQ**, **GCORRI**, **GCORRQ** and **IQCORR** parameter values are updated and **DC_BYP**, **GC_BYP**, **PH_BYP** bypasses are disabled. In case of calibration failure all chip parameters are restored to their state prior to calibration.

6.2 Receiver

Calibration requires user to configure receiver to working state and choose desired bandwidth in MHz. The calibration can be performed by using **CalibrateRx(float_type bandwidth_MHz)**, it automatically changes **MAC** parameter and restores it after procedure ends. If the calibration succeeds **EN_DCOFF_RXFE** is enabled and **DCOFFI**, **DCOFFQ**, **GCORRI**, **GCORRQ**, **IQCORR** parameter values are updated also **DC_BYP**, **GC_BYP**, **PH_BYP** bypasses are disabled. In case of calibration failure all chip parameters are restored to their state prior to calibration.

7

Filter tuning

7.1 Transmitter

Transmitter filters can be tuned by using **TuneTxFilter(TxFILTER filterType, float_type bandwidth_MHz)** function. This function can tune **TX_LADDER**, **TX_REALPOLE** and **TX_Highband** filters. It requires to select desired filter type and its bandwidth. In case of low band chain separate function should be used **TuneTxFilterLowBandChain(float_type ladder_bw_MHz, float_type realpole_bw_MHz)**, which takes two separate ladder and realpole filter bandwidth parameters. If desired bandwidth is not available for selected filter, function will return **LIBLMS7_FREQUENCY_OUT_OF_RANGE** status. In case of tuning failure, chip configuration will be restored to state prior to tuning procedure.

7.2 Receiver

Receiver filters can be tuned by using **TuneRxFilter(RxFILTER filterType, float_type bandwidth_MHz)** function. This function can tune **RX_TIA**, **RX_LPF_LOWBAND**, **RX_LPF_Highband** filters. It requires to select desired filter type and its bandwidth. If desired bandwidth is not available for selected filter, function will return **LIBLMS7_FREQUENCY_OUT_OF_RANGE** status. In case of tuning failure, chip configuration will be restored to state prior to tuning procedure.

8

Additional functionality

8.1 Interface frequency

To configure interface frequencies use `SetInterfaceFrequency(float_type cgen_freq_MHz, uint8_t interpolation, uint8_t decimation)` function. It requires to specify CGEN frequency in MHz, Tx interpolation (`HBI_OVR`) and Rx decimation (`HBD_OVR`) parameters. According to given parameters **MCLK1SRC**, **MCLK2SRC**, **RXTSPCLKA_DIV**, **TXTSPCLKA_DIV**, **RXDIVEN**, **TXDIVEN** parameters will be changed.