ML-DSO-API 2.0.5 API Manual

Contents

Introduction

In this document we explain the different functions of the DSO API.

Most functionality is demonstrated in details in the sample code release in the same package with the API library.

For any questions please email ghabre@multilaneinc.com and/or support@multilaneinc.com

List of Supported Measurements

a) TopMeasurement, BaseMeasurement, MinMeasurment, MaxMeasurment
b) RiseTimeEye, FallTimeEye
c) PeakToPeak
d) EyeAmplitude, VoltageAmplitude
e) Optical Measurements: OMA, AOP, ExtinctionRatio
f) EyeHeight, EyeHeightTop, EyeHeightBase
g) EyeWidth
h) JitterP2P, JitterRms
i) CrossingY, CrossingX
j) Mask Measurements: PointsPerUI, FailingPointsCount, MaskMargin
k) RJrms, RJrmsMinIntrinsic, RJppByBER, DJ
l) TotalJitterByBER, TotalJitterJ2, TotalJitterJ9, EyeWidthByBER
m) OneLevel, ZeroLevel, and for PAM4: ThreeLevel, TwoLevel
n) RiseTimePattern, FallTimePattern
o) PreEmphasisHeight, PreEmphasisWidth, DeEmphasisHeight, DeEmphasisWidth
p) DN, RNrms, RNpp
q) TotalNoiseByBER, EyeHeightByBER
r) SNR
s) VerticalEyeClosure
t) Pam4_AVupp, Pam4_AVmid, Pam4_AVlow
u) Pam4_Vupp, Pam4_Vmid, Pam4_Vlow
v) Pam4_Hupp, Pam4_Hmid, Pam4_Hlow
w) Pam4_VuppBER, Pam4_VmidBER, Pam4_VlowBER
x) Pam4_HuppBER, Pam4_HmidBER, Pam4_HlowBER

MLDSO-API Functions and Parameters

In this section we describe the various API functions and their parameters

Connect To DSO (required)

```
bool ConnectToDSO(string IP, bool isOpticalDevice, int opticalWavelength)

Routine Description:

 Connect to Board

Arguments:

       IP: The IP of the board to be connected.

Return Value:

 Returns true if the connection was successfully made
```

Configure Application (required)

```
bool ConfigureApplication(char* saveConfig, char* saveData, bool SaveStatistics, bool
SaveData, bool statisticsPrint, bool eyePrint)

Routine Description:

 Configure the Application settings

Arguments:

 saveConfig:         Configuration location folder where clock clkConf is saved
 saveData:           Captured data are saved in the specified location
 SaveStatistics:     Enable and disable statistics saving
 SaveData:           Enable and disable data saving
 statisticsPrint:    Enable and disable statistics print in console
 eyePrint:           Enable and disable eye print in console

Return Value:

 Returns true if ConfigureApplication was successfully executed
```

Configure DSO (required)

```
bool ConfigureDSO(double lineRateInput, double packetCountInput, int patternLengthInput,
bool wanderOn, bool precisionTimeBaseOn, bool normalModeInput, double clockInput,
double nrzWindowStartInput, double nrzWindowsEndInput,
```

```
bool configureDSO, int samplingFrequencyIndex, double attenuatorCompensation,
double totalJitterTargetBER, double deviceIntrinsicJitterFemtoSec, double dJCorrection,
double rJrmsCorrection, bool isFilterPjAtSource, int percentUIforLevels, double &
samplingFreq, E_SIGNAL_CODING signalCoding)
```

Routine Description:

This function configures the DSO for execution. After this function one on more executions can occur. This function rarely takes up to 2 minutes when executing it for the first time for a specific input frequency, from then on it would cache and reuse the settings. If it hangs more than 2 minutes please contact support

Arguments:

1. lineRateInput: Set linerate of the signal in Gbps
2. packetCountInput: Packet count need to be captured 1-512, Packet = 512 sample per packet
3. patternLengthInput: Signal pattern length, 1-PN13 if higher pattern measurements won't work but still do eye measurements = 50 for longer patterns would give best speed without pattern lock
4. wanderOn : Apply wander confiscation on the captured signal
5. precisionTimeBaseOn: Apply precision time baes on the signal, Remove RJ from the Signal
6. normalModeInput: Normal mode = true, Bypass mode = false
7. clockInput: Value of the input clock MHz
8. nrzWindowStartInput: Value of the NRZ Windows Start default(0.20) means RT&FT start on 20%
9. nrzWindowsEndInput: Value of the NRZ Windows End default(0.80) means RT&FT start on 80%
10. configureDSO: Default **false**. Apply the Linerate change on the signal or not, if you are not changing line rates it will improve performance significantly to set this as **false**
11. samplingFrequencyIndex: Selected sampling frequency index, Default(0)
12. attenuatorCompensation: in dB, Amplifies the result, commonly used to compensate for the effect of having an attenuator
13. totalJitterTargetBER: example: 1e-12 , this is the target BER to use when calculating Total Jitter, Total Noise, Eye Height as a function of BER
14. deviceIntrinsicJitterFemtoSec: default 350
15. dJCorrection: default 1, keep at 1, used only for troubleshooting
16. rJCorrection: default 1, keep at 1, used only for troubleshooting
17. isFilterPjAtSource: default is false, used only for troubleshooting and special cases as recommended by support
18. percentUIforLevels: default 20 for 20 % , this is the percentage to take around the eye center for measurements including levels, amplitudes, and eye heights. Some compliance tests specify percentages different than 20
19. & samplingFreq: This returns the samplingFrequency selected to be used by the DSO. A PLL algorithm runs when executing this function in order to configure how the sampling clock is to be generated from the input reference clock. This function may take up to 2 minutes when executing it for the first time for a specific input frequency, from then on it would cache and reuse the PLL settings. If it hangs more than 2 minutes please contact support
20. signalCoding: Specifies the encoding of the signal as NRZ or PAM4, from the enumerator E_SIGNAL_CODING. The enumerator values are either escNRZ or escPAM4. Defaults to NRZ

Return Value:

```
 Returns true if ConfigureDSO was successfully executed
```

Configure Eye Measurements (optional, all true by default)

```
bool ConfigureEyeMeasurements(
bool doTopMeasurements, bool doBaseMeasurements, bool doMin,
bool doMax, bool doRiseTime, bool doFallTime, bool doPeakToPeak,
bool doEyeAmplitude, bool doEyeHeight, bool doEyeWidth,
bool doCrossingY, bool doJitter, bool doSNR, bool doVEC)


Routine Description:

 Enable and Disable specific eye measurement

Arguments:

 doTopMeasurements: Enable Top measurement ...

SNR is signal to noise ratio in dB

VEC is vertical eye closure in dB

Return Value:

 Returns true if ConfigureEyeMeasurements was successfully executed
```

Configure Pattern Measurements (optional, all true by default)

```
bool ConfigurePatternMeasurements(
bool doRJ, bool doDJ, bool doRiseTimePatterm,
bool doFallTimePattern, bool doPreEmphasisHeight,
bool doPreEmphasisWidth, bool doDeEmphasisHeight,
bool doDeEmphasisWidth, bool doNoise, string restrictRiseTime, string restrictFallTime)



Routine Description:

 Enable and Disable specific pattern measurement

Arguments:

 doRJ: Enable RJ measurement ...

You may restrict pattern rise/fall tie measurement to occur only for specific bit
sequences in the pattern such as the rise time in 000001 or the fall times in 111110
sequences

Return Value:

 Returns true if ConfigurePatternMeasurements was successfully executed
```

Configure PAM 4 Measurements (optional, all true by default)

```
bool ConfigurePam4Measurements(
bool doVerticalEyeHeights, bool doHorizontalEyeWidths,
bool doVerticalEyeClosure, bool doVerticalEyeHeights_By_BER, bool
doHorizontalEyeWidths_By_BER)
```

Routine Description:

Enable and Disable specific PAM 4 measurements

Return Value:

Returns true if ConfigurePam4Measurements was successfully executed

Configure Mask Measurements (optional, mask disabled by default)

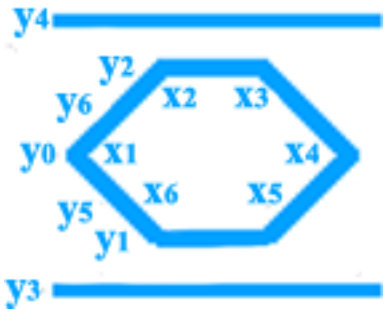
```
bool ConfigureMaskMeasurements(bool doMaxMargin, double targetFailingPointsHitRatio,
int maskMarginPercentage,double x1, double x2, double x3, double x4, double x5, double x6,
bool maskYAsPercentage, double y0, double y1, double y2, double y3, double y4, double y5,
double y6)
```

Routine Description:

Configures mask measurements

Arguments:

1. doMaxMargin: Apply max margin calculation for a target number of failing points or Hit Ratio
2. targetFailingPointsHitRatio: is > 1 is considered as the target number of failing points. If 1 then this variable will define the Hit Ratio as a percentage of the total number of points in one Unit Interval of the Eye.
3. maskYAsPercentage: Ys as percentage of the Eye Amplitude or as mV static Value. Usually optical signals standards are defined as percentage whereas electrical as static values
4. Xs & Ys please check below image for a better understanding. Also you may check the DSO GUI



Return Value:

Returns true if the ConfigureMaskMasurements was successfully made or not

Configure Filters (optional, Filters disabled by default)

This function will automatically enable PTB (Equally Spaced Sampling) if it was originally disabled

```
bool ConfigureFilters(int ptbResolution, string filtersFile)
```

Routine Description:

Configures Filtering

Arguments:

1. ptbResolution: An integer power of 2 (64, 128, 256) which should be defined less than NumberOfPackets * PatternLength / 512. Where NumberOfPackets and PatternLength are parameters provided to the Configure DSO function
2. The path to the filter file saved from the DSO GUI Filters Screen. Note that this file may contain more than one filter, but its execution at the end of the day will take the same performance requirement if it were one or more than one filter. Filters are usually defined in the GUI, and their effect is checked there in the GUI as well. Once the desired set of filters is decided, they are convolved together and used as one combined filter.

Return Value:

Returns true if the Configure Filters was successfully executed

Capture And Measure (required Always)

This is the main function for performing measurements and dumping results in files. All other required functions need only be executed once at the beginning, as long as the API instance remains unharmed (no exceptions thrown) this function can be executed over and over as much as needed without needing to re-execute any of the other required functions.

```
bool CaptureAndMeasure(int chIndex, int acquisitionCountInput, int accumulateNumPoints)
```

```
Routine Description:

 Capture data and measure signal on the ML-DSO

Arguments:

   1. chIndex:             Specify which channel need to be measured (Send 0 for
                           Single-Channel DSO)
   2. acquisitionCountInput: Default (1) Acquisition count, their average will be returned
   3. accumulateNumPoints: Default (0) to do only on capture. This is the
                           total number of points to accumulate per 2 UI,
                           which is the number of UIs that displays in the Eye diagram
                           in the GUI. The actual hardware Capture (And Number of points
                           in the Pattern diagram) would be half this amount

Return Value:

 Returns true if CaptureAndMeasure was successfully executed
```

Get Results or Statistics (optional)

```
string GetResults()
string GetResultsMin()
string GetResultsMax()
string GetResultsStd()
string GetResultsLatest()
```

```
Routine Description:

Gets capture results or statistics (if CaptureAndMeasure -> acquisitionCountInput > 1) in
a coma separated string format. An Enumerator E_MEAS is defined to facilitate accessing
the results. Also a helper function:
vector<double> ConvertStringToList(string stringToConvert, char characterSeparator)

Return Value:

string GetResults()                    Returns Average Results
string GetResultsMin()                 Returns Minimum Results
string GetResultsMax()                 Returns Maximum Results
string GetResultsStd()                 Returns Standard Deviation of different Results
string GetResultsLatest()              Returns Results of last acquisition
```

Read Eye Data (optional)

```
int ReadEyeData(vector<double>& xEyeValues, vector<double>& yEyeValues)

Routine Description:

 Return the Data captured in a vector of double, which can be used to draw the eye diagram

Arguments:

 xEyeValues: Returned array of X Eye Valued
 yEyeValues: Returned array of Y Eye Valued

Return Value:

 Returns data count for number of samples in the eye capture
```

Read Pattern Data (optional)

```
Routine Description:

 Return the Data captured in an vector of double

Arguments:

 xPatternValues: Returned array of X Pattern Valued
 yPatternValues: Returned array of Y Pattern Valued

Return Value:

 Returns data count

int ReadPatternData(vector<double>& xPatternValues, vector<double>& yPatternValues)
```

Special Purpose Functions (Don't use unless instructed by MultiLane)

```
bool AccessBoardRegister(UInt16 Read_Write, UInt16 Index, UInt16 Reg, UInt16 *Data)

bool WriteScopeCalibration(char* IP, int chIndex, bool isOptical, int opticalWaveLength,
double A, double B)

bool ReadScopeCalibration(char* IP, int chIndex, bool isOptical, int opticalWaveLength,
double &A, double &B)

bool DisconnectFromDSO()  // Not Needed. All DSO Connections are closed Automatically
```

Usual Flow of Required Functions

In this section we describe the usual flow of the required functions

1- Instantiate a DSO_Wrapper object where applicable (C#, C++, C), otherwise an ML_DSO_API object. Keep reusing this instance for the same DSO channel to avoid the need to reconfigure things every time.
2- ConnectToDSO
3- ConfigureApplication
4- ConfigureDSO

Loop as much as needed:

5- CaptureAndMeasure
CaptureAndMeasure will print out the results; also it will dump results in .csv files if enabled through ConfigureApplication.
6- GetResults (optional): If it is not enough to dump the results in files, you may use GetResults, GetResultsMin, GetResultsMax, GetResultsLatest, or GetResultsStd to retrieve the latest results and statistics programmatically, and use them for processing. Use the E_MEAS enumerator to understand which value each csv column represents.

Sample Code

Sample code is usually provided with the API as well and may show more features

```
bool ML_DSO_Measure()
{
	fprintf(stderr, "#########################################\n");
	fprintf(stderr, "===>> ML-DSO starts to take measurement    \n");
	fprintf(stderr, "#########################################\n");

	const int NUM_CHANNELS = 4;

	string WORK_DIR = "/home/DSO-API/";

	MLDSOAPI ml4025API[NUM_CHANNELS];
	bool isSimulation[NUM_CHANNELS] = { false, false, false, false };
	std::string iP[NUM_CHANNELS] = { "172.16.102.18", "172.16.102.18", "172.16.102.25",
"172.16.102.25" };
	int channels[NUM_CHANNELS] = { 0, 1, 0, 1 };
	double lineRateInput[NUM_CHANNELS] = { 25.78125, 25.78125, 25.78125, 25.78125 };
	double packetCountInput[NUM_CHANNELS] = { 128, 128, 128, 128 };
	int patternLengthInput[NUM_CHANNELS] = { 511, 511, 511, 511 };
	bool wanderOn[NUM_CHANNELS] = { false, false, false, false };
	bool precisionTimeBaseOn[NUM_CHANNELS] = { false, false, false, false };
	bool normalModeInput[NUM_CHANNELS] = { true, true, true, true };
	double clockInput[NUM_CHANNELS] = { 161.1328125, 161.1328125, 161.1328125, 161.1328125
};
	double nrzWindowStartInput[NUM_CHANNELS] = { 0.2, 0.2, 0.2, 0.2 };
	double nrzWindowsEndInput[NUM_CHANNELS] = { 0.8, 0.8, 0.8, 0.8 };
	bool configureDSO[NUM_CHANNELS] = { false, false, false, false };
	int samplingFrequencyIndex[NUM_CHANNELS] = { 0, 0, 0, 0 };
	double attenuatorCompensation[NUM_CHANNELS] = { 0, 0, 0, 0 };
	double totalJitterTargetBER[NUM_CHANNELS] = { 1e-12, 1e-12, 1e-12, 1e-12 };
	double deviceIntrinsicJitterFemtoSec[NUM_CHANNELS] = { 350, 350, 350, 350 };

	bool isfiltering[NUM_CHANNELS] = { enableFiltering != 0, enableFiltering != 0,
enableFiltering != 0, enableFiltering != 0 };
	std::string filterFiles[NUM_CHANNELS] = {
		WORK_DIR + "ctle_4_rate25.78125_ptb128.fltr",
		WORK_DIR + "ctle_4_rate25.78125_ptb128.fltr",
		WORK_DIR + "ctle_4_rate25.78125_ptb128.fltr",
		WORK_DIR + "ctle_4_rate25.78125_ptb128.fltr" }; // where .fltr file was saved
from DSO GUI


	vector<string> simFiles[NUM_CHANNELS] = { vector<string>(), vector<string>(),
vector<string>(), vector<string>() };

	for (int k = 0; k < NUM_CHANNELS; k++)
	{
		// [OPTIONAL]
		if (isSimulation[k])
		{
			// [OPTIONAL]
			simFiles[k].push_back(WORK_DIR + "Simulation.dso");
		}

		int connectResult = ml4025API[k].ConnectToDSO(iP[k], false, 0, simFiles[k]);

		if (connectResult == 1)
		{
			cout << " Connected Successfully " << endl;
		}
		else
		{
			cout << " Failed to Connect " << endl;
			return false;
		}
```

```
// Constructing data dump folder name to contain DSO IP and Channel
string runDir = WORK_DIR + "Run/";
std::stringstream chan;
chan << channels[k];
string resultsDir = runDir + "Results_" + iP[k] + "_" + chan.str() + "/";


cout << ml4025API[k].ConfigureApplication((char*)runDir.c_str(), (char*)resultsDir.c_str(),
        // Draw eye in command line
        true,
        // Dump Eye and Pattern Data
        false,
        // Display statistics
        true,
        // Dump Statistic data
        false) << endl;

// [OPTIONAL]
cout << ml4025API[k].ConfigureDSO(lineRateInput[k], packetCountInput[k],
patternLengthInput[k], wanderOn[k], precisionTimeBaseOn[k], normalModeInput[k],
        clockInput[k], nrzWindowStartInput[k], nrzWindowsEndInput[k], configureDSO[k],
samplingFrequencyIndex[k], attenuatorCompensation[k], totalJitterTargetBER[k],
deviceIntrinsicJitterFemtoSec[k], 1, 1, false) << endl;


// [OPTIONAL]
if (isfiltering[k])
{
        // [OPTIONAL]
        cout << ml4025API[k].ConfigureFilters(128, filterFiles[k]) << endl;
}

// [OPTIONAL]
ml4025API[k].ConfigureEyeMeasurements(
        // doTopMeasurements,
        true,
        // doBaseMeasurements,
        true,
        // doMin,
        true,
        // doMax,
        true,
        // doRiseTime,
        true,
        // doFallTime,
        true,
        // doPeakToPeak,
        true,
        // doEyeAmplitude,
        true,
        // doEyeHeight,
        true,
        // doEyeWidth,
        true,
        // doCrossingY,
        true,
        // doJitter,
        true,
        // doSNR
        true,
        // doVerticalEyeClosure
        true);
```

```
        // [OPTIONAL]
        ml4025API[k].ConfigurePatternMeasurements(
                // doRJ,
                true,
                // doDJ,
                true,
                // doRiseTimePatterm,
                true,
                // doFallTimePattern,
                true,
                // doPreEmphasisHeight,
                true,
                // doPreEmphasisWidth,
                true,
                // doDeEmphasisHeight,
                true,
                // doDeEmphasisWidth,
                true,
                // doNoise
                true,
                // Restrict Pattern Rise Time measurement to below bits
                "000001",
                // Restrict Pattern Fall Time measurement to below bits
                "111110");

        // [OPTIONAL]
        cout << ml4025API[k].ConfigureMaskMeasurements(
                true // doMaxMargin
                , 0 // targetFailingPointsHitRatio
                , 0 // maskMarginPercentage
                , 0.31 // x1
                , 0.5 // x2
                , 0.5 // x3
                , 0.69 // x4
                , 0 // x5
                , 0 // x6
                , false // maskYAsPercentage
                , 0 // y0
                , -42.5 // y1
                , 42.5 // y2
                , -600 // y3
                , 600 // y4
                , 0 // y5
                , 0 // y6
                ) << endl;

}
```

```
        for (int k = 0; k < NUM_CHANNELS; k++)
        {
                cout << ml4025API[k].CaptureAndMeasure(channels[k], numAcquisitions, 0) <<
endl;

                // [OPTIONAL]
                vector<double> results = ConvertStringToList(ml4025API[k].GetResults(), ',');

                // [OPTIONAL]
                double eyeHeight = results[emeasEyeHeight];

                // [OPTIONAL]
                cout << iP[k] << " Channel:" << channels[k] << ", Results:" << endl
                        << "topMeasurment:\t" << results[emeasTopMeasurement] << endl
                        << "baseMeasurment:\t" << results[1] << endl
                        << "minMeasurment:\t" << results[2] << endl
                        << "maxMeasurment:\t" << results[3] << endl
                        << "riseTimeEye:\t" << results[4] << endl
                        << "fallTimeEye:\t" << results[5] << endl
                        << "riseTimeEyeStartX:\t" << results[6] << endl
                        << "riseTimeEyeEndX:\t" << results[7] << endl
                        << "fallTimeEyeStartX:\t" << results[8] << endl
                        << "fallTimeEyeEndX:\t" << results[9] << endl
                        << "peakToPeak:\t" << results[10] << endl
                        << "eyeAmplitude:\t" << results[11] << endl
                        << "eyeHeight:\t" << eyeHeight << endl
                        << "eyeHeightTop:\t" << results[13] << endl
                        << "eyeHeightBase:\t" << results[14] << endl
                        << "eyeWidth:\t" << results[15] << endl
                        << "jitterP2P:\t" << results[16] << endl
                        << "sNR:\t" << results[emeasSNR] << endl
                        << "jitterRms:\t" << results[emeasJitterRms] << endl;
        }


        return true;
}
```


North America
48521 Warm Springs Blvd. Suite 310
Fremont, CA 94539
USA
+1 510 573 6388

Worldwide
Houmal Technology Park
Askarieh Main Road
Houmal, Lebanon
+961 5 941 668

Asia
14F-5/ Rm.5, 14F., No 295
Sec.2, Guangfu Rd. East Dist.,
Hsinchu City 300, Taiwan (R.O.C)
+886 3 5744 591