MODBUSCLIENT DLL API
Rev. 4.9.3

Revision History

Revision	Functions	Comment

3.0	`startDataClientPLC`
	`updateThicknessPLC`
	`getThicknessUnitPLC`
	`stopMeasurementPLC`
	`startMeasurementPLC`
	`setRecipePLC`	

3.5	`getRawData`
	`getRawDataFloat`
	`calculateData`	

4.0	`getIntTimeMicrosec`
	`getThicknessFloat`
	`updateCalculationPropertiesPLC`	

4.5	`getMemoryStatus`
	`getRecipesList`
	`getFilmstacksList`
	`getConnectionAddress`
	`setMeasurementIntervalPLC`
	`init_custom`
	Bugs fixed:
	1.Setting wrong measurement recipe raises exception. This exception was persisting (not cleared) when the correct recipe is set.
	2. setFilmstack function stopped working after previous Modbus updates (adding PLC connector)
	3. Reading IP and port from Modbus.properties correctly.
		May 28,2024

4.6	`getDllVersion()` `getParameters()`	**July 24, 2024**
4.7	`setParameter()` `getCalibrationData()`	**September 15, 2024**
4.8	`startRawMeasurementPLC` `stopRawMeasurementPLC`	**October 30, 2024**
4.9	`startIntensityMeasurementPLC()` `stopIntensityMeasurementPLC()`	**March 8, 2025**
4.9.1	`1.Changed calculateData signature to include measurement recipe.` `2.refactored dll to have jvm instance and classes persist between the calls`	**March 23,2025**

83 Pine Hill Rd. Southborough, MA 01772
Phone +1.617.388.6832 Fax. +1.508.858.5473
email: info@semiconsoft.com http://www.semiconsoft.com

4.9.2	Added 3 new functions: getRawMeasurement() getIntensityMeasurement() calculateFromIntensity()	**April 18,2025**
4.9.3	getReferenceSignal() setReferenceSignal() freeMemory() Updated ownership, initialization, thread attachment, and binary format documentation.	October 26, 2026

83 Pine Hill Rd. Southborough, MA 01772
Phone +1.617.388.6832 Fax. +1.508.858.5473
email: info@semiconsoft.com http://www.semiconsoft.com

Introduction

ModbusClient.dll provide a remote interface for MProbe measurement systems using Modbus style TCP=IP protocol. It facilitates communication between a client and the TFCompanion software, enabling remote measurement control..
ModbusClient.dll can be located on the same computer as TFCompanioo software (that is connected to MProbe system) or on any other computer on the network.

ModbusClient.dll is a C library that interfaces with Java implementation of the Modbus Client (located in Modbus.jar). ModbusClient.dll is communicating (via Modbus.jar) with TFCompanion's build-in ModbusServer software**.**

Installation & Configuration

Prerequisites

1. **Java installation:** Java should be installed on the computer.
   If you are using a different computer on a network (not the one where TFCompanion is installed) – check that java is installed.
   You can check it by typing: **java –version** at the command prompt. Java version should be 1.8 or 1.9 (earlier versions may be incompatible). Versions later are not tested
2. System Path Configuration**.**
   ModbusClient.dll and **jvm.dll** should be added to the system path
   (jvm.dll is located in java installation directory e.g.
   C:\Program Files\Java\jre1.8\bin\server)
   Note. The log file (Modbus.log) will be saved in the working directory of the program that calls **ModbusClient.dll**

Library compatibility

For DLL revision 4.9.3, use Modbus.jar version 3.9.5 build 102626 and tfcomp.jar build 102626. Place these files alongside the DLL with the other required JAR files.

The structure of the installation directory:

```
<home_directory>/
                    <executable program that uses ModbusClientDLL..dll >
                    Modbus.properties (optional)
              library/
                        --- ModbusClientDLL..dll
                        --- Modbus.jar
                        --- tflib.jar
                        --- log4j.jar
                        --- comm..jar
                        --- tfcomp.jar  (required since rev 4.6)
            log/
                ---log_config.properties
```

<home_directory> is the directory where executable (calling) program (the program that uses ModbusClientDLL.dll) is located e.g. ModbusExample.exe

Returned buffers and memory management

a. Every non-NULL pointer returned by this API, except getDllVersion(), is an independent DLL allocation. Release it exactly once with freeMemory(), including response buffers from measure(), checkStatus(), and setters. Do not use the caller's free(), delete[], or another allocator.
b. Buffers remain valid across later API calls and disconnect() until released. Release all buffers before the DLL is unloaded. Input buffers remain caller-owned. getDllVersion() returns a borrowed, read-only string: do not modify it or pass it to freeMemory().

```
void freeMemory(void* memory);
```

freeMemory(NULL) is allowed. A non-NULL pointer must be a buffer returned by this DLL and must not have been freed already.

Functions with an int* length output require a non-NULL pointer. On failure they return NULL and set *length to 0; an empty array also returns NULL with length 0. Lengths count bytes for char arrays and elements for float arrays.

getRecipesList(), getFilmstacksList(), and getConnectionAddress() append a NUL terminator that is excluded from *length. Other byte arrays are binary data. Decode binary numeric fields in the documented byte order; float* results are native C float arrays. Strings passed to JNI use modified UTF-8.

Example of releasing a returned buffer

After initialization, connection, and a successful measurement, retrieve and release the raw-data array as follows:

```
int length = 0;
float* data = getRawDataFloat(&length);
if (data != NULL) {
    /* Use data[0] through data[length - 1] here. */
}
freeMemory(data);  /* Also safe when data is NULL. */
data = NULL;       /* Do not access or free it again. */
```

Apply the same rule to a response buffer: signed char* response = measure(); must eventually be paired with freeMemory(response), even when the response is read through convenience getters.

On JNI failure, boolean functions return 0; PLC command and calculation functions return 1; numeric and status getters fall back to 0. A zero status alone does not establish success. JNI exceptions are reported to stderr and cleared before returning to C.

API Details

1. Initialization.

Call init() or init_custom() before other client calls. A successful call initializes the JVM and a single process-lifetime client. Keep the DLL loaded for the lifetime of the process.

```
/**
  @param ip - non-NULL server address string; use "localhost" for the same
computer.
  init() uses port 502. init_custom() accepts an explicit port.
  Pass the string "0" to read the IP address from Modbus.properties. To read the
configured port too, use init_custom("0", 0). Do not pass a NULL pointer.
  @return 0 on success; a negative JNI error on failure. NULL ip returns -6
(JNI_EINVAL); an already initialized client returns -5 (JNI_EEXIST).
*/
long  init(char* ip);

/**
Alternative to init(char* ip), this function allows to specify both IP and port
*/
long  init_custom(char* ip, int port)
```

Initialization failure never replaces an existing client. If JVM creation succeeds but client construction fails, a retry reuses that JVM. disconnect() closes only the connection; use connectClient() to reconnect, without reinitializing.

2. Connection management

```
/*
Connects the client to the address and port selected during initialization; init()
uses port 502.
If you are using firewall or virus protection program – please make sure that
port#502 is open.
By default connection timeout is 3000 ms. If response from the server is not
received during this time – connection is terminated. The client will try to
reestablish connection 2 times before giving up.
To set custom timeout use setConnectionTimeout(int time_ms)
method – this method need to be called BEFORE connectClient()
otherwise it will have not effect.

return boolean. 1 - true: connection is successful, 0 (false) - problem during
connection

*/
 unsigned char connectClient();

/*
Closes the server connection. The initialized client and JVM remain available for
reconnection.
*/
void disconnect();
/** Returns 1 when connected, or 0 when disconnected or on JNI failure. */
unsigned char isClientConnected();
```

83 Pine Hill Rd. Southborough, MA 01772
Phone +1.617.388.6832 Fax. +1.508.858.5473
email: info@semiconsoft.com http://www.semiconsoft.com

```
/*
Sets connection timeout in milliseconds. Default timeout id 3000ms
Setting timeout to 0 –means connection will never timout.
IMPORTANT: This method should be called BEFORE calling connectClient();
Otherwise it will have no effect
*/
void setConnectionTimeout(int time_ms);

/**
@return current timout in milliseconds
*/
int getConnectionTimeout();

/**

@return the active server address and port; NULL when disconnected.
@param length – required output; byte count excluding the NUL terminator.
Example format  192.168.1.88:7
*/

 char* getConnectionAddress(int* length);
```

3. Measurement Configuration.

Measurement parameters need to be set before the first measurement is executed. These parameters are remembered and used for all future measurements. If needed, parameters can be reset at any time.

```
/**
     * Sets measurement parameters in the output Hashtable to prepare request.
     * The content of the Hashtable is maintained and used for all transactions.
     * @param channel - channel number (0 or 1)
     * @param waferID - Sample ID (up to 40 bytes length)
     * @param recipeName - name of the measurement or calibration recipe (up to 10
     *  bytes long). If recipe does NOT have filmstack attached – Filmstack needs
     *  to be set separately before calling measure
     *@see measure()
     *@see setFilmstack()
     * @param expectedThickness - OPTIONAL PARAMETER (0 - to ignore). Value of the
         Expected thickness - resets the thickness of the currently used filmstack
     * @param intTime - OPTIONAL PARAMETER (0 - to ignore). Resets integration
     time in the specified recipe measurement recipe
     * @return true (1) if successful
     */

unsigned char setMeasurement(int channel,
                             char* waferID,
                             char* recipeName,
                             int expectedThickness,
                             short intTime)
```

83 Pine Hill Rd. Southborough, MA 01772
Phone +1.617.388.6832 Fax. +1.508.858.5473
email: info@semiconsoft.com http://www.semiconsoft.com

```
/**
     * A convenience method to change measurement recipe used in transaction.
     * @ see setMeasurement method.
     * The data is stored in the hashtable and will be used to prepare next
       response
     * @param recipeName the name of measurement recipe
     * @param channel measurement channel number
     * @return true (1)- recipe set successfully, false (0) - problem setting
        recipe.
     */

 unsigned char setRecipe(int channel, char* recipeName)

/**
* Sets new Filmstack
* If recipe already have filmstack attached - it is replaced with the new
* filmstack
* @param matName - filmstack name, length up to 40 char
* @return a binary server-response buffer, not a boolean success flag.
* Release a non-NULL response using freeMemory().
*/
signed char* setFilmstack(char* matName);




/**
     * Convenience method to switch between standard measurement and measurement
       w/ diagnostics Can be applied after setMeasurement was called but before
        measure() function is called.
     * @see setMeasurement method
     * @param diag true (1) to enable measurement with diagnostics, false (0)
        standard measurement
     * standard measurement is the default
        *@return  true (1) if successful, false (0) if problem
     */
unsigned char setDiagnostics(unsigned char flag)
```

83 Pine Hill Rd. Southborough, MA 01772
Phone +1.617.388.6832 Fax. +1.508.858.5473
email: info@semiconsoft.com http://www.semiconsoft.com

4. Performing Measurement

```
/**
     * Execute measurement or calibration based on the data set in the output
       Hashtable defined by setMeasurement and setRecipe
     * @return response from the server. This response can be parsed directly
       using Response(Read) table spec or
     * use convenience methods that parse response information.
     * @see getExceptionCode(), getThickness()
      Measurement call is blocking (it is waiting for the response from the
      server). The default  timeout is set to 3 sec. (see @setConnectionTimeout())
      If response is not received – client will disconnect and try connect again
      (it will repeat it 3 times before giving up)

 */
signed char* measure()
```

After the measure() function is successfully completed – a set of values become available on the client and can retrieved using following functions

```
Following functions are the client calls – the data is already transmitted from
the server after the measure() function call. These functions need to be called
after the measure() call and before any other server call
/**
Parse server response to extract thickness data
@param channel – the measurement channel
@return  - thickness in Angstroms
*/

long getThickness(int channel)

/**
* Parse server response to extract GOF data
@param channel: measurement channel
@return fit MSE (goodness of fit), distinct from thickness confidence interval.
*/
double getGOF(int channel)


/**
* Parse server response to extract confidence interval data
@param channel: measurement channel
@return thickness confidence interval in Angstroms. Read after measurement or
calculation, before another server request replaces the response.
*/
double getThicknessError(int channel)

/*
available in diagnostic mode

@param channel: measurement channel
@return actual signal level (intensity) in % of the total range
*/

short getSignalLevel(int channel)
```

```
/**
Parse server response to extract integration time available in diagnostic mode

Return actual integration time used during the measurement
@param channel channel number
@return integration time in ms
*/
short getIntegrationTime(int channel)
```

Additional functions are available to request the data from the server.

```
/**
Added in Rev. 4
@ret thickness value as a float in default unit (unit defined in TFCompanion
configuration)
*/
float getThicknessFloat();

/**
Added in rev.4
@ret integration time in microseconds

*/
int getIntTimeMicrosec();
```

5. Checking for exceptions and warnings

```
/**
@return warning code
NO_WARNING=0
LOW_SIGNAL=1 (low light intensity)
HIGH_SIGNAL 2 (intensity maybe too high - saturation)
LOW_MEMORY 3 (low memory available to the program)
*/
signed char getWarning()
```

```
/**
     * Parse server response to extract exception code during transaction
        (measurement/calibration)
     * Can be called after response was received
     * @return 0 - no exception, 1,2 or 3 - general Modbus errors (communication/
        format problem),
     * 4 - unknown sensor error, 5 - data acquisition problem, 6 - calculation
        problem, 7 - generic system exception, 8 - request-content error (for
example, an invalid or missing recipe).
     */
signed char getExceptionCode()

/*
      This function can be called concurrently.
     * Parse server response to check system status
     * Can be called after response was received
     * @return 0 - ready/idle,1 - measuring, 2 - calculating, 3 - exception
      state,4 - busy/initializing
*/
signed char getSystemStatus()
/** Requests current server status; replaces the cached server response. */
signed char* checkStatus();
```

Read getSystemStatus(), getWarning(), and getExceptionCode() after the response. Release the returned response buffer with freeMemory().

```
/** Returns the type of the latest request; 0 is also the JNI failure fallback. */
signed char getRequestType();
```

6. Modifying Filmstack

```
/**
* Adds a new layer on top of the filmstack with default thickness 10A
* Set thickness of this layer as a calcualted/display parameter and disable
* other calculated parameters
* @param matName the name of the material
*/
signed char* addLayer(char* matName);

/**
* Removes the layer with the specified material
*@param matName the name of the material in the layer to be removed
*/
signed char* removeLayer(char* matName);
```

7. Checking intensity

Check Maximum Signal.

```
This method allows to check maximum signal from the system before doing actual
measurement. A full spectrum is acquired and a maximum value of the intensity
is determined. Averaging of the 10 spectra is done to have accurate data.
In case of the system with Flush lamp – setStrobe() command should be send
first to activate synchronization pulses.
```

In case of all other systems, integration time should be checked to make sure it is the same as will be used during measurement (signal value will depend directly on integration time)

```
/**
@return maximum intensity as counts of a 16 bit ADC from 0 to 65500
*/
unsigned short getMaxIntensity()
```

Set Intensity

This method allows to set intensity of the light source.

```
/**
@param value – value in the range of 0 to 100, 100 – maximum intensity
@return a response from the server  (byte[])
*/
signed char* setIntensity(unsigned char value)

/** action: 1 starts the strobe; 0 stops it. delay: milliseconds from
 * acquisition start to strobe. Release the response with freeMemory(). */
signed char* setStrobe(unsigned char action, short delay);
```

Library compatibility for this revision is listed under Installation & Configuration.
Use Modbus.jar version 3.9.5 build 102626 with tfcomp.jar build 102626.

8. Retrieving measured and calculated Reflectance/Transmittance data

These functions are not compliant with Modbus specification because of the length of the message (Modbus message is limited to 256 bytes)
During each measurement the data is cached. The cache is replaced after each measurement. These methods return the results of the last measurement taken

```
/**
* @ret a byte array of the raw reflectance and calculated data
* as a 4 byte floats: wavelength, measured value, calculated value
* The last 4 bytes is a GOF (goodness of fit value)
* @ret length - the length of the byte array returned
*/
 char* getRawData(int* length);


/**
*
@ret a n array of floats same as @see getRawData
@ret length - the length of the array for convenience
*/
float* getRawDataFloat(int* length);
```

8.1. Measure and calculate using Reflectance/Transmittance/intensity data

```
/**
* Returns a malloc-allocated float array with measurement data.
 * Caller must release the returned pointer using freeMemory().
* Measurement of the reflectance or transmittance spectra (depending on the
recipe)
* without calculation
@ret float array of the measured data with same format as @ see
startRawMeasurementPLC
* Wavelength, data.... wavelength, data
* @param length - the length of the returned float[]
* @param recipeName - the name of the measurement recipe (recipe should match
the calibration)
*/
 float* getRawMeasurement(char* recipeName,int* length );


/**
* Measurement of the Intensity spectrum based on the specified measurement
recipe
* @ret float array of the measured data with same format as @see
startIntensityMeasurementPLC
* Wavelength, data.... wavelength, data, integration time in ms
* @param length - the length of the returned float[]
* @param recipeName - the name of the measurement recipe
*/

float* getIntensityMeasurement(char* recipeName,int* length );




/**
Added Modbus rev. 3.5 modified rev. 4.9.1
This function can be used after measurement is done and raw data is received
@ see getRawDataFloat
Or it can used independently in conjuction with continuous RawReflectance
measurement
@see startRawMeasurementPLC
param float* - an array of wavelengths and measured reflectance as floats
(32bit)
Each measurements point includes wavelength and reflectance without
separators: lam[0]reflectance[0]....lam[n]reflectance[n]
@param length - the length of the array
@param mesRecipeName  recipe name, should have filmstack attached
@return 0 - success, 1 - failure
After function executed successfully, can use getRawDataFloat to retrieve data.
@see getRawDataFloat(int* length)
*/
unsigned char calculateData(float* data, int length, char* mesRecipeName);
```

```
/**
 *Perform calculation using intensity data. Provided intensity data is
converted to reflectance or transmittance
 *based on the measurement parameter identified in the measurement recipe.
Calculation is performed usign filmstack
 attached to the measurement recipe.
 The results canh be retrieved the same way after standard measurement @see
measure()
 @param data - an array of float in the format  received from @see
getIntensityMeasurement() or
 function 85 of the Dataclient output i.e. wavelength,
value....wavelength,value, integration time(ms)
 @param length - the length of the data array
 @param recipeName - the name of the measurement recipe
 @ret 0 - success, 1 - failure
*/

unsigned char calculateFromIntensity(float* data, int length, char*
recipeName);
```

9. Auxiliary functions

```
/**

 @return  free memory and used memory in MB of java on server
 8 bytes (two float numbers) are returned
*/
unsigned char* getMemoryStatus();


/*
Added Rev. 4.5
@return a list of available measurement recipes names
comma separated as char*
@param length the length of the char* returned
*/
 char* getRecipesList(int* length);

/*
@return a list of available filmstacks names
comma separated as char*
@param length the length of the char* returned
*/

char* getFilmstacksList(int* length, char* match);

/**
@return borrowed, read-only DLL version and build date/time; do not free it.
*/
char* getDllVersion()
```

```
/*
Added Rev. 4.6
*@param length - the length of returned array
*@return binary calculated-parameter records in the following format:
* Each record is name*value*CIvalue*; * is byte 42, including the final
separator.
* Records repeat without padding:
name1*value1*CIvalue1*name2*value2*CIvalue2*
* name is the parameter name. Read fixed-width binary fields by size,
* not by searching for separators. value is a 4-byte big-endian float.
* CIvalue is the 1-sigma confidence interval in 4-byte scientific notation:
* two big-endian signed shorts a and b, with CIvalue = a * 10^b.
* length counts bytes. The buffer is not NUL-terminated; use freeMemory().
*/
char* getParameters(int* length);

/**
*@see getParameters
*@param name - the name of the parameter in the filmstack, e.g. 1_Thickness
is ther thickness of the first layer
*@param value- the value of the parameter that will be set
*/

unsigned char* setParameter(char* name, float value);
```

By default each call attaches the calling thread to the JVM when needed and detaches an attachment it owns on return. startSingleThread() retains that thread's attachment until stopSingleThread(). Pair these calls on the same thread before it exits. Repeated starts are idempotent; an attachment already owned by the caller is never detached.

```
void startSingleThread();

void stopSingleThread();
```

```
/**
*@param recipe the measurement recipe corresponding to the calibration
* @length is a convinience parameter passed by the client to know the length
of the returned floats array
* @return calibration data as floats (without separators) in the following
sequence:
* wavelength, intensity_counts,dark counts, reference_reflection_fraction
* ********
* wavelength,intensity_counts,dark counts,reference_reflection_fraction
* integration_time(in ms)
*/

float* getCalibrationData(char* recipe_name, int* length);
```

Calibration loading

```
/** fileName: calibration file name (up to 20 characters). Use an empty
 * string or "*" for the default. Release the response with freeMemory(). */
signed char* loadCalibration(char* fileName);
```

83 Pine Hill Rd. Southborough, MA 01772
Phone +1.617.388.6832 Fax. +1.508.858.5473
email: info@semiconsoft.com http://www.semiconsoft.com

9.1 Reference signal functions

Use these functions to transfer calibration reference data or a current reference update between reference and measurement channels. Both functions require a non-NULL recipeName and int* length output. All input memory remains owned by the caller.

Retrieve reference signal data

```
unsigned char* getReferenceSignal(const char* recipeName,
    unsigned char base, const char* path, int* length);
```

base: nonzero selects the reference acquired during calibration; zero selects the current reference update.

path: optional file path on the TFCompanion server. Pass NULL to retrieve data without saving it to a file.

Returns binary reference data in the Java library's wire format. The DLL adds no NUL terminator and performs no byte-order conversion. *length is the byte count. On failure or an empty response, returns NULL with *length set to 0. Release a non-NULL buffer with freeMemory().

Set reference signal data

```
unsigned char* setReferenceSignal(const char* recipeName,
    unsigned char ref, unsigned char mode,
    const unsigned char* data, int dataLength,
    const char* path, int* length);
```

ref: nonzero selects the initial reference; zero selects a reference update.

mode nonzero: copy dataLength bytes from data; path is ignored. dataLength must be nonnegative, and data may be NULL only when dataLength is 0. Supply the binary format returned by getReferenceSignal(); do not pass a native float array or use strlen() to determine its length.

mode zero: load from a required, nonempty file path on the TFCompanion server. data and dataLength are ignored.

Returns a binary server-response buffer, not a success flag. *length counts response bytes. On failure or an empty response, returns NULL with *length set to 0. Check the server response/status for operation success and release a non-NULL response using freeMemory().

String arguments use JNI modified UTF-8. Required libraries: Modbus.jar version 3.9.5 build 102626 and tfcomp.jar build 102626.

10. Pass-through functions that are executed by the PLC library

```
**
* Following functions require PLC.jar on the classpath.
* They are passthrough function that are executed by the PLC library
* Note.
*  a. Measurement recipe need to be set to PLC using
*     setRecipePLC() function.
*  b. Opposite to other Modbus functions, these functions return 0 if success and
*     1 – is failed.

/**
* Starts build-in Data Client of the PLC server
* Data client connects to an external Data Server.
* The type of the expected  Data server is defined in the PLC.properties
* The options are:
* - analog output is directed to the DAC (in this case IP and port corresponds to
the FPGA board connected to DAC)
* - TCP the TCP data client is starts and expects a TCP data server to connect to.
* - UDP the UDP data client starts and expects a UDP data server to send data (not
current implemented)
* Data server should be running on IP address and port specified in the call
 @param ip IP address of the Data Server
 @param port is a port of the Data server
  @ret 0 - success, 01- failed
*/
unsigned char startDataClientPLC(char* ip, int port);

/**
Added Rev. 3.0
@param name - the name of the measurement recipe
The measurement recipe is required to have a filmstack attached
This recipe is set in the PLC library and is independent of any
recipe set in Modbus server
@ret 0 - success, 1 - failed
*/
unsigned char setRecipePLC(char* name);


/**
Added Rev. 3.0
Start continuous measurement using PLC server library
@param delay - delay between the measurements in ms.
It overrides the delay set in the PLC.properties file
@ret 0 - success, 1 - failed
*/

unsigned char startMeasurementPLC(int delay);
```

```
/**
Added Rev. 3.0

Stops continuous measurement
@see startMeasurementPLC
@param cond - currently ignored
The program will allow the currently running
calculation to complete before stopping (normal termination)
@ret 0 - success, 1 - failed
*/
unsigned char stopMeasurementPLC(unsigned char cond);

/**
@ret currently used thickness unit
0- Angstroms
1- nm
2 - microns
The default unit is set in TFCompanion configuration
*/

unsigned char getThicknessUnitPLC();

/**
Added Rev. 3.0
Updated the thickenss value in the filmstack used for calculations
in the PLC library
@param paraName is the name of the parameter. 1_Thickness - first layer thickness
@param thickness - the value of the new thickness
@param unit unit used for the thickness param
@ret 0 if success, 1 - failed
*/

unsigned char  updateThicknessPLC(char* paramName, float thickness, int
unit);

/**
* Updates calculation algorithm in the PLC library during the measurement
* This function should be used during the measurement. it will fail if
measurement is not started
*
@param XML element <Calculation_Conditions> form the measurment recipe
       that defines the calculation algorithm and related parameters
@ret 0 if success, 1 - failed
*/

unsigned char  updateCalculationPropertiesPLC(char* xmlString);
```

Example of XML element for FFT calculation

```
"<Calculation_Conditions>"
"<algorithm>Thick Film FFT< / algorithm>"
"<fft_level>0.04</fft_level>"
"<FFT_interface>false</FFT_interface>"
<Calc_Preprocess_remove_trend>true</Calc_Preprocess_remove_trend>"
"<CALC_ACCURACY>0</CALC_ACCURACY>"
"<Calc_total_thickness>false</Calc_total_thickness>"
"</Calculation_Conditions>";
```

```
/**
Sets interval between the measurements
This function can be used when measurements are active.
If the measurement is not running - the function will fail
@param delay interval between the measurements in ms
*/
unsigned char setMeasurementIntervalPLC(int delay);

/**
Added Rev. 4.8
Start continuous raw measurement (as defined in the recipe) using PLC server
library
This functions pre-requisites: Measurement recipe needs to be set and Data
client started
@see setRecipePLC
@see startDataClientPLC
@param delay - delay between the measurements in ms.
Setting delay=0 will remove any delay between the measurements
It overrides the delay set in the PLC.properties file
@ret 0 if success, 1 - failed
*/
unsigned char startRawMeasurementPLC(int delay);


/**
Stops continuous raw measurement
@see startRawMeasurementPLC
The measurement thread will stop after the current data acquisition is
completed
@ret 0 if success, 1 - failed
*/
unsigned char stopRawMeasurementPLC();
```

```
/**
Start continuous intensity measurement (as defined in the recipe) using PLC
server library
Measurement recipe should define fixed integration time
This function pre-requisites: Measurement recipe needs to be set and Data
client started
The data is sent to the DataServer (function 85) - please see Data server
specification
@see setRecipePLC
@see startDataClientPLC
@param delay - delay between the measurements in ms.
It overrides the delay set in the PLC.properties file
@ret 0 if success, 1 - failed
*/
 unsigned char startIntensityMeasurementPLC(int delay);

/**
Stops contnuous Intensity measurement
@see startIntensityMeasurementPLC
The measurement thread will stop after the current data acquisition is
completed
@ret 0 if success, 1 - failed
*/
unsigned char stopIntensityMeasurementPLC();
```

83 Pine Hill Rd. Southborough, MA 01772
Phone +1.617.388.6832 Fax. +1.508.858.5473
email: info@semiconsoft.com http://www.semiconsoft.com