TRIANGLE MICROWORKS, INC.

# Source Code Library Implementation Guide

Version 3.26.0

December 15, 2019

# Chapter 1 Introduction

Congratulations on your purchase of a Triangle MicroWorks, Inc. (TMW) communication protocol Source Code Library (SCL). This product is a member of a family of communication protocol libraries supported by Triangle MicroWorks, Inc. These libraries were designed to be flexible, easy to use, and easily ported to a wide variety of hardware and software platforms. For these reasons, the Triangle MicroWorks, Inc. Source Code Libraries will provide an excellent solution to your communication protocol requirements.

## 1.1 Manual Conventions and Abbreviations

The following conventions and abbreviations are used throughout this manual.

TMW – Triangle MicroWorks, Inc.

SCL – Source Code Library

Test Harness - The Triangle MicroWorks, Inc.  Communication Protocol Test Harness

<application> - Name of the sample application program. (DNPMaster, DNPSlave, 104Master, 104Slave, etc.)

# Chapter 2 Getting Started

The purpose of this Guide is to provide users a quick step-by-step guide to port the TMW sample application and SCL to the target platform

This section will highlight each step of the porting process and provide a very high-level description of what is required. If more information is required, please refer to the SCL User Manual.

The Source Code Libraries are shipped with sample application(s) and build environment that can be compiled and run on a Windows or Linux host without modification. If the target environment is Windows, simply unpack and build the SCL for Windows. If the target OS is Linux or another OS, TMW recommends initially building the SCL and the sample application on a Linux host as a base line for the porting effort.

## 2.1 Unpack and Install the SCL

Each TMW SCL is delivered as a self-extracting ZIP file. When delivered via email, the ZIP file is encrypted, and you will have to enter the password (usually delivered by Fax) in order to extract the SCL.

Extract the SCL by running the self-extracting ZIP file. The SCL files will be installed in the desired target directory (<*installdir*>). TMW recommends retaining the directory structure of the SCL as it is supplied.

## 2.2 Build the SCL

The first step in porting the SCL should be to build it before making any modifications. Once installed, all SCL source code files reside in the subdirectories under <*installdir*>*/tmwscl*. The sample application resides in the <*installdir*>*/application* directory.

### 2.2.1 Building on Windows

The SCL is shipped with solution and project files (*.sln, *.vcxproj) that are configured for Visual Studio 2010. If you are using a newer version of Visual Studio the project files can be updated automatically.

The solution file for this application resides in <*installdir*>/*application*.sln  (e.g DNPSlave.sln). The source code for the sample application resides in a subdirectory under <*installdir*>*/application*  (e.g., DNPSlave) . Load the solution file and build the example application.

### 2.2.2 Build on Linux (GNU Make & gcc)

The SCL is shipped with GNU Makefiles that are configured to use gcc to build the SCL and the sample application. The default build target builds the sample application and the SCL. Enter the **make** command from the <*installdir*> to perform the build. The application will be linked in the <*installdir*>/bin directory and can be executed by simply entering the command: **./bin**/<*application*>.

## 2.3 Run the sample application

The sample application is setup with a default configuration that will allow it to communicate with Test Harness. The sample application and Test Harness can be configured to communicate serially or over a TCP connection.

To configure the sample application to use serial communication, it will be necessary to update the default configuration. Edit the sample application <installdir>/<application>/<application>.cpp. Update the initialization of the boolean **useSerial** variable to true. Also search for the serial configuration variables that begin with **IOCnfg.targ232** and verify that they are set correctly. Ensure that **IOCnfg.targ232.portName** is set for the correct serial interface.

Alternatively, the sample application can be configured to use a TCP connection. If working with a Slave sample application, no configuration changes should be necessary. If working with a Master sample application, the sample application's IP Address configuration will have to be updated to specify the address of the Slave device (Test Harness). This address is specified in the variable **IOCnfg.targTCP.ipAddress.**

If any configuration changes were made, rebuild the sample application.

Run the sample application. Then start the Test Harness and configure a channel to communicate with the sample application.  Use this configuration to become familiar with the SCL and the Test Harness's features.

## 2.4 Porting the Target Layer

### 2.4.1 Target Layer selection

The SCL is packaged with 3 target layers: Linux, Windows, and a sample template implementation. These target layers are implemented in the <installdir>/tmwscl/tmwtarg directory as **LinIoTarg**, **winiotarg**, and **SampleIoTarg**. The default builds for Linux & Windows are configured such that they compile and link with the appropriate target layer. The GNU Makefiles are set up such that the sample target layer can be compiled and linked against when the command **make config=sample** is entered.

If the target system is using Linux or Windows, simply use the existing Linux or Windows target layer and build environment to begin the porting effort and the remainder of this section can be skipped. If the target system is using another operating system, then the sample target template has been provided as a starting point for the target layer port. The sample template target layer can be compiled and linked with the SCL with no operating system dependencies.

Once a target layer has been selected for port, all the files in its directory should be included in the build. The directories for the other two target layers should not be used and can be removed. If using one of the existing target layers to start the port, TMW recommends renaming the target layer directory and the files it contains appropriately. For example SampleIoTarg, it should be renamed VxwIoTarg if the port is for the VxWorks operating system..

### 2.4.2 Target Build

The build environment should be configured to include the target layer directory in its default include path. Ensure the target build compiles all of the source files in each of SCL subdirectories (excluding the unused target layers) and the sample application.

Once configured, build the SCL and sample application to determine if any compiler issues need to be resolved.

The file *'tmwscl/utils/tmwtypes.h'* includes definitions for data types that are used throughout the SCL. The type definitions are based on stdint.h. Most target platforms will require few, if any, changes to this file. However, this file may need to be modified if the target compiler does not support the C99 standard.

If any changes to this file are required, they will usually be limited to the `/* Type Definitions */` section.

It is important that type definitions for TMWTYPES_ULONG and TMWTYPES_LONG are configured to specify 32-bit values. The SCL can run on 32 or 64-bit targets, but expects these data types to specify 32 bit types regardless of the underlying processor architecture.

### 2.4.3 Dynamic Memory Allocation Support

The sample target layer template uses dynamic memory allocation to allocate its data structures. If the target environment cannot support calls to 'malloc' and 'free' they will need to be replaced with a static allocation scheme for any required target layer allocations. The SCL can support either configuration.  See the SCL's User Guide, Memory Allocation section for details.

### 2.4.4 Thread Support

The SCL can be configured to run in a multithreaded environment or a single execution context. The SCL is shipped with thread support enabled. Both the Linux and Windows target layers are written to run in a multithreaded environment. The Linux target layer can also support running in a single execution context. If the target OS can support threads, TMW recommends configuring the SCL to support threads. If it cannot, channels must be polled periodically for data which will add latency.
If the target OS can support threads:

- Edit the twmscl/utils/tmwcnfg.h file and set **TMWCNFG_SUPPORT_THREADS** to **TMWDEFS_TRUE**.
- Edit the tmwscl/tmwtarg/XXXIoTarg/tmwtargos.h file:
  - Define the **TMW_ThreadCreate** macro such that when invoked, it will start a thread with the given arguments.
  - Optionally define the **TMWDEFS_RESOURCE_LOCK** macro as a pointer to a semaphore. The default void* definition is sufficient for the SCL, but if this macro is defined, it can help with debugging as the debugger will be able to dereference the semaphore pointer and display its contents instead of simply displaying a void pointer.
- Edit the tmwscl/tmwtarg/XXXIoTarg/tmwtarg.c file:
  - Update the implementation of semaphore functions, *tmwtarg__lockInit*, *tmwtarg__lockSection*, *tmwtarg__unlockSection*, *tmwtarg__lockDelete*, and optionally *tmwtarg_lockShare*. Make sure the *tmwtarg__lockInit* function initializes the semaphore such that it can be taken recursively.
  - Update the implementation of *tmwtarg_sleep*.

If the OS cannot support threads:

- Edit the twmscl/utils/tmwcnfg.h file and set **TMWCNFG_SUPPORT_THREADS** to **TMWDEFS_FALSE**.
- Ensure that the channel configuration parameter, polledMode is always set to true.
- Ensure the application calls the function *tmwappl_checkForInput* frequently to check for received data.

## 2.4.5 Time Support

### *2.4.5.1 Monolithic Millisecond Timer*

The SCL needs a continuously running millisecond timer. This support is implemented in tmwscl/tmwtarg/XXXIoTarg/tmwtarg.c file.

- Update the function *tmwtarg_getMSTime* such that it returns a 32-bit unsigned millisecond count.

### *2.4.5.2 System Time*

The SCL's default configuration gets the current system time to timestamp diagnostic messages in its log, data changes in the simulated database, the slave samples use it to determine when to call xxx_addEvent functions. The default database implementation for slave 101,103,104, and DNP devices set the system time in response to a clock synchronization request. Support for these functions is optional and implemented in tmwscl/tmwtarg/XXXIoTarg/tmwtarg.c file. Edit this file:

- Update the function *tmwtarg_getDateTime* such that is returns the system date and time into a **TMWDTIME** structure.
- Update the function *tmwtarg_setDateTime* such that it updates the system date and time with the value provided in a **TMWDTIME** structure if the target application should update the system time in response to a time synchronization request.

## 2.4.6 Timer Support

The SCL can be configured with a single timer queue or a timer queue per channel. If the application needs to support a large number of channels, it is more efficient to run with a timer queue per channel. If this is the case:

- Edit the twmscl/utils/tmwcnfg.h file and set **TMWCNFG_MULTIPLE_TIMER_QS** to **TMWDEFS_TRUE**.
- If the target OS supports threading, the *_channelThread* implementation provided, supports the multiple channel timer callback. If not, it will be necessary to implement the functions *tmwtarg_initMultiTimer*, *tmwtarg_setMultiTimer*, and *tmwtarg_deleteMultiTimer* to support the multitimer callback functionality.

Otherwise, to support a single timer queue:

- Implement the functions *tmwtarg_startTimer* and *tmwtarg_cancelTimer*.

## 2.4.7 Channel Support

The sample target layer has support for serial and TCP channels. Determine which support is needed for the target environment. By default, both are enabled. If only one is required, then edit the tmwscl/tmwtarg/XXXIoTarg/tmwtargcnfg.h file:
If only TCP channels are required:

- Set **TMWTARG_SUPPORT_232** to **TMWDEFS_FALSE**.
- If working with a DNP application, then **TMWTARG_SUPPORT_UDP** should be set to **TMWDEFS_TRUE.**
- Delete the files XXX232.h and XXX232.c.

- Remove the #include XXX232.h from all target layer files.

If only serial channels are required:

- Set **TMWTARG_SUPPORT_TCP** to **TMWDEFS_FALSE**.
- Delete the files XXXtcp.h and XXXtcp.c.
- Remove the #include XXXtcp.h from all target layer files.

The header files sample232.h and sampleTCP.h each have a minimal channel structure (**SERIAL_IO_CHANNEL** and **TCP_IO_CHANNEL**) defined. Any OS specific data required to support channel operations should be added to these structures. These header files also contain  the function prototypes required to support the channel. The source files sample232.c and sampleTCP.c each have template implementations for  these functions. Each of these functions will need to be implemented before the channel will operate with the library. The Linux target layer can be used as a reference implementation of each function.

### 2.4.8 Verify the Sample Application on the Target

Once the target layer port has been completed, the sample application should run on the target device. Run the sample application on the target and run the Test Harness. Update the configuration and make any changes necessary to account for the differences from the target device and the Linux host where the sample application was run in Section 2.3 such as the IP address or serial port name.

## 2.5 Modify SCL Configuration Files

### 2.5.1 SCL General configuration

The file <*installdir*>/tmwscl/utils/tmwcnfg.h is used to configure the SCL. The values that are shipped with the SCL target a general-purpose configuration with support for threads, dynamic memory, and diagnostics enabled. If the target device is memory constrained, it may be necessary to disable some of the configuration parameters by setting the values to **TMWDEFS_FALSE**. If possible, its best to leave the parameter **TMWCNFG_SUPPORT_DIAG** set to **TMWDEFS_TRUE** during development so that information about any errors is displayed on the console.

Each protocol's general configuration file is listed in Table 2. These files are used to specify the maximum sizes for various arrays used by the TMW SCL. In most cases, the default values can be used for the initial porting effort. However, they can be reduced if working on a memory constrained target.

**Table 1. Protocol Configuration Files**

| **Protocol** | **Configuration File** |
|---|---|
| DNP Master | mdnpcnfg.h |
| DNP Slave | sdnpcnfg.h |
| IEC 60870-5-1-101 Master | m14cnfg.h |
| IEC 60870-5-1-101 Slave | s14cnfg.h |
| IEC 60870-5-102 Master | m102cnfg.h |
| IEC 60870-5-102 Slave | s102cnfg.h |
| IEC 60870-6-103 Master | m13cnfg.h |
| IEC 60870-5-103 Slave | s13cnfg.h |
| IEC 60870-5-104 Master | m14cnfg.h |
| IEC 60870-6-104 Slave | s14cnfg.h |
| Modbus Master | mmbcnfg.h |
| Modbus Slave | smbcnfg.h |

Rebuild the SCL after updating the configuration to ensure there are no conflict.

### 2.5.2 Turn off the Simulated Database

The default configuration for the SCL sample programs is to use the simulated database. With the simulated database enabled, the samples will behave more like a real device. However, the purpose of the simulated database is to provide a simple working example to get started with. It is not designed to be memory efficient nor does TMW attempt to minimize changes to it from release to release. **TMW strongly discourages integrating the simulated database into target applications.**

Edit the general configuration file described in the previous section and set the parameter **TMWCNFG_USE_SIMULATED_DB** to **TMWDEFS_FALSE.** Once this change is made, the sample application will no longer be configured to use the simulated database. This will save significant memory resources and reduce the effort required to integrate future releases. Remove the file tmwsim.c and the protocol's Simulated Database Operations File shown in **Table 2** from the build.

**Table 1. Simulated Database Files**

| **Protocol** | **Simulated Database Header Files** | **Simulated Database Operations File** |
|---|---|---|
| DNP Master | mdnpsim.h | mdnpsim.c |
| DNP Slave | sdnpsim.h | sdnpsim.c |
| IEC 60870-5-1-101 Master | m14sim.h | m14sim.c |
| IEC 60870-5-1-101 Slave | s14sim.h | s14sim.c |
| IEC 60870-5-102 Master | m2sim.h | m2sim.c |
| IEC 60870-5-102 Slave | s2sim.h | s2sim.c |
| IEC 60870-6-103 Master | m3sim.h | m3sim.c |
| IEC 60870-5-103 Slave | s3sim.h | s3sim.c |
| IEC 60870-5-104 Master | m14sim.h | s14sim.c |
| IEC 60870-6-104 Slave | s14sim.h | s14sim.c |
| Modbus Master | mmbsim.h | mmbsim.c |
| Modbus Slave | smbsim.h | smbsim.c |

.

## 2.6 Integrate the SCL with your Database

The interface between the SCL and the device database is defined in the database interface definition file defined in **Table 3**. This file defines which data types are supported as well as the prototypes of the database functions that will need to be implemented.

**Table 2: Database Files**

| Protocol | Database Interface Definition | Database Impementation File |
|---|---|---|
| MDNP | mdnpdata.h | mdnpdata.c |
| SDNP | sdnpdata.h | sdnpdata.c |
| IEC 60870-5-101 Master | m14data.h | m14data.c |
| IEC 60870-5-101 Slave | s14data.h | s14data.c |
| IEC 60870-5-102 Master | m2data.h | m2data.c |
| IEC 60870-5-102 Slave | s2data.h | s2data.c |
| IEC 60870-5-103 Master | s3data.h | s3data.c |
| IEC 60870-5-103 Slave | s3data.h | s3data.c |
| IEC 60870-5-104 Master | m14data.h | m14data.c |
| IEC 60870-5-104 Slave | s14data.h | s14data.c |
| Modbus Master | mmbdata.h | mmbdata.c |
| Modbus Slave | smbdata.h | smbdata.c |

When the SCL is shipped, the database interface definition file is configured to enable common data types for the protocol. This file should be modified to enable only data types needed by the target application. Disabling data types that are not needed for the target application will reduce memory consumption and the number of functions that will have to be implemented in the database file.

Start the database implantation by modifying the 'tmwscl/*<family>/<scl>data.c'* file. With the simulated database disabled, each function in this file will execute the stanza delineated with the comment /* Put target code here */. Other than the initialization function described in the next paragraph, the default database implementation functions are written such that they return non-fatal errors so functions supporting each data type can be integrated and tested individually.

The <scl>data_init function must be updated to return a non-null value otherwise the sample application will interpret it as a failure and exit. This function will be the first database function called by the library and should be used to perform any database initialization that may be required before the library performs calls to retrieve data. The library is implemented such that it will pass the pointer returned by this function to the database functions to retrieve data. If the target application does not require such a pointer, any non-zero value can be returned.

For each data type supported by the application, there are function(s) in this file to retrieve or store the data. Each of these functions will need to be updated such that they access or store the target's data instead simply returning a failure. The database interface file listed in **Table 3** describes how each of these database function should be implemented.

## 2.7 Issue Commands (Master Sessions)

For Master sessions, commands are issued by calling routines in the files *'tmwscl/<family>/<scl>brm.h/c'*. Each of the routines in these files returns a handle that can be used to identify the request.

When the request is completed, the SCL will call a user callback function. The callback function arguments include the request and pointers to the transmitted request and the received response.

Information on protocol-specific commands are given in the protocol-specific document.

More details on issuing requests can be found in the master specific section of the User's Guide.

## 2.8 Add Support for Change Events (Slave Sessions)

For Slave sessions, the SCL can be configured to periodically scan for change events. Alternately, events can be generated directly by calling the appropriate *'xxx_addEvent'* function, where *'xxx'* specifies the type of event being generated.

Enabling SCL scanning for change events is a configuration option that is specified when opening the session or sector.

This topic will be described in more detail in the slave specific section of the User's Guide.

## 2.9 Test your Implementation

To test your implementation, you will need a system or application to act as the 'opposite' end of the communication link. For example, if you are implementing a Slave device, you will need a corresponding Master.

Triangle MicroWorks provides a flexible, scriptable test harness that can be used for this purpose. The TMW Communication Protocol Test Harness is a Windows application that acts as a simple Master or Slave device and can be programmed with an automated test sequence through a scripting capability.

The protocol analyzer output from the Test Harness allows you to verify the proper operation of the device you are testing. It also allows you to generate a reference protocol analyzer output file that can be used for comparison to later tests to verify only intended modifications have occurred.

For more information please see the Triangle MicroWorks, Inc. web site at *'http://www.TriangleMicroWorks.com'*.