# User Manual - Modbus Slave Protocol Library - 'C' Source Code

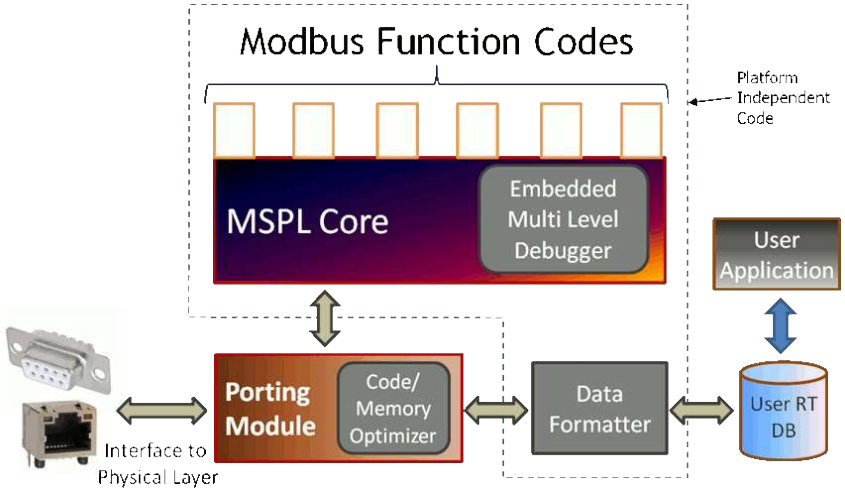
# Architecture

**Key architectural principles:-**

- Simplicity - to reduce code size
- Maximum portability - Strict compliance to ANSI 'C' standards
- Robust - only static memory allocation
- Sparing use of code and memory
- Modular, scalable and configurable - easy to maintain
- Easy to debug

# MSPL Block Schematic

Modbus Slave Libaray : Components, Organization and Interconnections

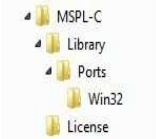



Components of Library:

| S.No. | Module | Functionality | File Name |
|---|---|---|---|
| **1** | MSPL Core | Frame Parsing<br>Packet Generation<br>Deploy Modbus Functions<br>Multi-level debugger | MSPL_C.c |
| **2** | Formatter | Data formatter. Modbus data types converted to:-<br>Short Integer<br>Integer<br>Float<br>String | CSPL_Utils.c |
| **3** | Porting Module | Links source code library to physical device and user application. | MSPL_Userlf.c<br>*This file modified by user* |

## Directory Structure

Folders within MSPL-C package:-




| Folder Name | Contents | Remarks |
|---|---|---|
| **Library** | Source files of MSPL-C | The files in this folder have the user definable hook functions left empty. |
| **License** | license agreement for the version of the library purchased | The license agreement has a unique license number which must be used in all correspondences with Colway Solutions regarding this library. |
| **Ports** | Ports of MSPL-C to Win32 and any other platform you requested. | The Win32 port can be found in "*Ports\Win32"* folder. This port contains project files to compile the source in MS Visual Studio 2008. If you requested for any other ports in addition to Win32, a relevant folder will also be included. |

## Files

The Modbus Slave Protocol Library contains the following 'C' source files:

| File Type | Filename | Contents | Engineer Modifies ? |
|---|---|---|---|
| **'C' Source** | MSPL_C.c | Modbus communication protocol stack | No |
| | CSPL_Utils.c | Library utility functions. Used by application too.<br>Formatter | No |
| | MSPL_Userlf.c | Platform dependent functions implemented by user "stubs" to receive platform dependent code. Refer to Win32 port for example. | Yes. |
| **'C' header** | MSPL_C.h | Header file for MSPL_C.c | No |
| | CSPL_Utils.h | Header file for CSPL_Utils.c | No |
| | CSPL_MbDefs.h | Colway Solutions type and symbol definitions for maximum portability.<br>CSPL_U16<br>CSPL_U132<br>etc. | Yes. Review and change for specific target |
| | MSPL_Userlf.h | Default values for all parameters. Refer to Win32 port for example | Yes. Extensive modification to complete port |

**Add MSPL files to your project**
After creating your project in the IDE of your platform, you must add all the files above into this project and if required explicitly configure all the above source files to be included in the build process.

## Hooks and Macros

**Hooks:**

- The porting of MSPL-C to a new platform is accomplished by means of defining hook functions.
- The hook functions are left unimplemented in the library
- Hook functions need to be implemented for porting the library

**Macros:**

- 'C' macros created using #define pre-processor statement
- Control conditional inclusion or exclusion of portions of the library code
- Define values for configuration parameters

## Porting

The following steps are required to port the Modbus Slave Protocol Library to your hardware and software environment.

**Step 1.** Add MSPL-C files to your project

**Step 2.** Define the Endian Architecture of your platform

**Step 3.** Select Modbus framing type (RTU or TCP)

**Step 4.** Glue MSPL-C to the physical interface of your platform

**Step 5.** Glue MSPL-C to the your application's database

**Step 6.** Configure diagnostics

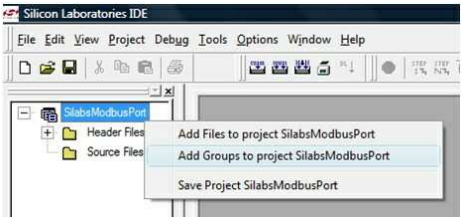
**Step 7.** Optimise MSPL-C

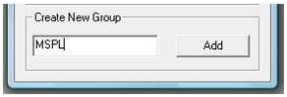
**Step 8.** Build and test your port with the supplied Tester

## Add Source Code to your Project

The first step in using MSPL-C is to add its source files to your project. The procedure for this step differs from one compiler or IDE to other. The following section describes this procedure with relevant screen shots for the Silicon Laboratories IDE. Procedure for other IDE's will be similar.

i.

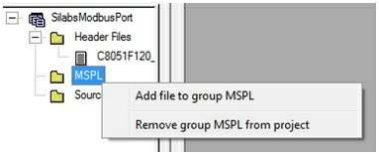


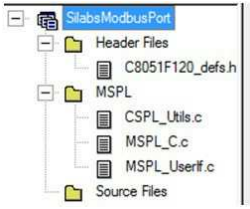




Create a new group called "MSPL" by right-clicking on the project name and choosing "Add Groups to project <proj name>" as shown below. Note that this step is optional.

ii. Right-click the mouse on the MSPL group created above. If the above step was skipped, right-click on any other group to which you intend to add MSPL-C files. Click on item "Add file to group <group name>". A File Open dialog box appears.

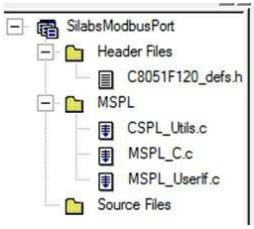
iii.




Browse to the folder containing the MSPL-C files and select all .c files. Click "Open".

iv. Press and hold the CTRL key and select all .c MSPL files. Right-click and choose "Add to build". This step is necessary to include the MSPL-C files in the compilation and build process.



# Set Endian Architecture

Modbus follows the Big Endian byte ordering system. Therefore the byte ordering has to be reversed if the Modbus library is deployed on a Little Endian processor. The library has a macro ENDIAN_STYLE, used to set the correct Endian characteristic.

**Steps**

a) Open file MSPL_UserIf.h

b) Locate the definition of macro *ENDIAN_STYLE*

c) If your platform is Little Endian, change the above macro's value to *LITTLE_ENDIAN*. If it is Big Endian, change the macro's value to *BIG_ENDIAN*. The modified line should look like this:

```
#define ENDIAN_STYLE LITTLE_ENDIAN/* for Little Endian */
#define ENDIAN_STYLE BIG_ENDIAN/* for Big Endian */
```

d) Rebuild your project and test.

**Notes**

- The utility functions provided by the Formatter (e.g. *MSPL_ShortIntsToBuffer*) are "Endian-aware" - they are programmed check and ensure that transfers from interpreted data types to raw buffers conform to Endianess of the platform.
- If you use your own code for such transfers, remember to address the issue of Endianess. Raw data in a Modbus packet is always in Big Endian format.
- To know the Endianess of your platform, refer to the User Manual of your processor.
- If you are unsure of the Endianess of your platform, a simple technique to determine this is to create a 'C' program with an *unsigned short int* variable (16-bit) and store the value 0xABCD in it:

```
unsigned short int testVar = 0xABCD;
```

Then debug this program and see the memory contents (using a Memory Dump or Memory Watch window) at the location of this variable. If you find 0xAB stored first and then 0xCD, you have a Big Endian system, else you have a Small Endian system.

## More About Endianness

Endianness is the byte (and sometimes bit) ordering used to represent some kind of data.
Also referred to as *byte order*.

For example a 'C' variable of data type *float* consists of four bytes. There are variations in storage sequence of these four bytes among different systems.

Endianness is crucial in communication systems implementation. Need to ensure that data reaches destination in the correct byte order.

Two most commonly used byte ordering systems are:

- *Big Endian.* Most significant byte of data unit is stored first in memory followed by the rest in descending order of significance. Motorola 68000 and PowerPC are examples of processors that adopt Big Endian byte ordering.

- *Little Endian*. The least significant byte of data unit is stored first in memory followed by the rest in ascending order of significance. Examples of such processors are Intel x86 and Z80.

**Note:** Most modern computer processors agree on bit ordering inside individual bytes. The library therefore has no provision for manipulating bit ordering.

## Select Modbus Framing Type(RTU or TCP)

The library supports two modes of Modbus communication, Modbus RTU and Modbus TCP. This can be set *at compile time* by setting the value of the MODBUS_MODE macro.

**Steps**

a) Open file MSPL_UserIf.h
b) Locate the definition of macro *MODBUS_MODE*
c) To configure the library to run in Modbus TCP mode, change the above macro's value to *MODBUS_TCP*. To set it to Modbus RTU mode, change the macro's value to *MODBUS_RTU*. The modified line should look like this:

```
#define MODBUS_MODE MODBUS_TCP /* for TCP communications */
#define MODBUS_MODE MODBUS_RTU /* for RTU communications */
```

d) Rebuild your project and test.

**Notes**

- Since this is a compile time setting, the mode cannot be changed dynamically at run time.
- Only one Modbus mode can be enabled at a time.

## Glue MSPL to Device Interface

A communication channel has to be set up between physical device and the Modbus library in order to receive Modbus request packets and transmit response packets.
The Modbus standard provides allows users to choose their own communication channel. Modbus compliant software is therefore unaware of the characteristics of particular communication channels.
Therefore the library provides a set of unimplemented (i.e. empty) hook functions that can be glued to the real interface functions of your communication channel.
The hook functions cover the four communication operations.

| S.No. | Channel Operation | Hook Function | Porting Notes |
|---|---|---|---|
| **1** | Open Port | MSPL_OpenPort | **i.** Use this function to open and configure communication channel<br>**ii.** User application must call this function once for every channel supported by the device<br>**iii.** A unique channel identification number is passed as an argument to this function.<br>**iv.** Device driver API usually returns a path identifier or handle to the channel being opened. This is required in subsequent operations: read, write and close. Please ensure that your program stores this identifier. See Win32 port implementation as an example. |
| **2** | Read from channel | MSPL_ReadPort | **i.** Library calls this function to read data from communication channel<br>**ii.** Function typically calls device driver's "Read" API<br>**iii.** A unique channel number is passed as an argument to identify the channel.<br>**iv.** Caution: Blocking calls to device driver API's in this function will block execution of MSPL-C as well as the application code that is calling the library. |
| **3.** | Write to channel | MSPL_WritePort | **i.** Library calls this function to transmit data on communication channel<br>**ii.** Function typically calls device driver's "Write" API<br>**iii.** A unique channel number is passed as an argument to identify the channel.<br>**iv.** Caution: Blocking calls to device driver API's in this function will block execution of MSPL-C as well as the application code that is calling the library. |
| **4.** | Close Port | MSPL_ClosePort | **i.** Use this function to close communication channel<br>**ii.** User application calls this function when no Modbus communication is required<br>**iii.** A unique channel number is passed as an argument to identify the channel. |

Google

## Supporting Modbus communication on multiple channels

MSPL-C supports simultaneous Modbus communication on multiple channels.
This is achieved by dedicating a set of all global variables to each communication channel.
Each channel now operates upon its own private dataset. In effect the library can be used to create *virtual Modbus devices*.
The macro MAX_NETWORKS is used to set the maximum number of channels required in the implementation.
Set this at compile time because the library uses static memory allocation.

Google
Online User Manual - MSPL

## Supporting multiple Modbus TCP connections in MSPL-C

A Modbus TCP slave application listens for connection requests on the Modbus port no. and accepts multiple connections. This requires a dedicated thread or task to listen for connection requests and accept them while another set of dedicated threads and tasks handle further communication with connected masters.

In MSPL-C, each connected socket is considered as a channel of communication. There are three approaches you can take to support for multiple Modbus TCP connections.

i. A dedicated thread for listening to connection requests and accepting them. Another set of threads to do further communication with connected clients (i.e. one thread per connected client). The connection handling thread can block waiting for new connection requests in this case.

ii. A dedicated thread for listening to connection requests and accepting them. Another thread (i.e. just one thread) to do further communication with connected clients. In this case too, the connection handling thread can block waiting for new connection requests.

iii. On platforms that do not have a facility for multitasking, a single thread (i.e. the main thread) will have to do the task of both listening for new connection requests as well as handling communication with connected clients. Obviously, the connection handling code cannot block waiting for new connection requests. Instead such code should only query the underlying TCP/IP stack to enquire if a new connection request has come in. If yes, it must be accepted and execution must continue to the code that handles further Modbus communication with the connected clients.

The Win32 port of MSPL-C demonstrates how to support multiple Modbus TCP connections simultaneously.

# Glue MSPL-C to application and database

## Glue library to Application

The library handles the task of framing and de-framing Modbus messages. The data within the messages are supplied by respective application programs. The library encapsulates this data as per Modbus framing rules and transmits it to the recipient.

Using the functions Read Coil and Write Coil to illustrate.

- Read Coil: In response to the Read Coil command, the application program running on the slave will supply data to the library. The library will frame the data in accordance to Modbus framing rules and send it to the master, completing the transaction.
- Write Coil: The application running on the master supplies the data to the library. The library encapsulates the data in the right frames and forwards the framed message to the slave program, which executes the command.

Two functions in the MSPL_UserIf.c file facilitate the interface between the library and your application and database.

- MSPL_ReadUserData: Called by MSPL-C when it receives a "read" type of Modbus request.
- MSPL_WriteUserData: Called by MSPL-C when it receives a "write" type of Modbus request.

***User is responsible for implementing these interface functions.***

These two functions present a well defined interface that is fully documented in this manual. The library supplied to you contains dummy implementations of these functions with no code within.

Please refer to sample Win32 port for a complete reference.

## Glue library to Simulated Database

A simulated database forms part of the library supplied. It has a few variables of all the data types supported by the library.
Use this database as a first step to get the library working on your platform. This exercise will assist in integrating the library with the application's database.

The database is created at the beginning of the MSPL_UserIf.c file and contains the following data elements:

| S.No. | Data Element | Associated Modbus Data Type | Number of Arrays | Memory Address |
|---|---|---|---|---|
| **1** | CSPL_U8 (single byte) | Coils, Discrete Inputs | 2 | 0000 to 0015 (16 items) |
| **2** | CSPL_U16 (two byte) | Holding and Input Registers | 2 | 0000 to 0010 (10 items) |
| **3** | Demonstrate mapping register to long integer | Float | 1 | 0020 to 0029 (5 floats using 10 registers) |
| **4** | Demonstrate mapping register to long integer | Long Integer - 4 bytes | 1 | 0040 to 0049 (5 long integers using 10 registers) |
| **5** | Demonstrate mapping register to string | Strings | 1 | 0060 to 0063 (8 characters that can hold a 'C' string of max length 7 using 4 registers) |

The interface functions in the MSPL_UserIf.c file operate upon this simulated database.
After testing with this database, you may replace it with your own.Â  Modify the interface functions to operate on your database.

## Using the Data Formatter to map 'C' data types to Modbus

MSPL-C provides you an extension to the Modbus specifications by supplying a set of functions in file *CSPL_Utils.c* that map the low level Modbus types (bits and words) to high level 'C' data types (floats, integers and strings) with due consideration to the ENDIAN format of your platform.

There are two categories of functions:

- Functions that convert an array of raw data bytes as received via Modbus to an array of higher level 'C' data type. They are usually called in *MSPL_WriteUserData* to interpret the raw Modbus data as per the corresponding higher level 'C' datatype of the user database.
- Functions that convert an array of some higher level 'C' data type into an array of raw data bytes that can be transmitted via Modbus. They are usually called in *MSPL_ReadUserData* to provide the library with user data in a Modbus compliant format.

Following is a brief description of each function:

| Function name | Description |
|---|---|
| **CSPL_PackBits** | This function *bit-packs* a destination buffer with bit status information provided in a source buffer. The source buffer is expected to contain bit status (i.e a value of 0 or 1) in one byte per bit. This data is bit-packed as 8-bits per byte in the destination buffer. For e.g. if the input buffer is of this type:<br>CSPL_U8 srcBuffer[] = {0,1,1,0,1,1,0,0}<br>then the destination buffer's first element will be stuffed with the following value - 0x6C; |
| **CSPL_UnPackBits** | This function extracts bit status information from the source buffer by unpacking the bits and copies it to the destination buffer. The source buffer is expected to have bit packed data with one byte holding the status of 8 bits. For e.g. if the if the input buffer is of this type:<br>CSPL_U8 srcBuffer[]={0xF1, 0xAB}<br>then this function will unpack this data to create the following in the destination buffer<br>CSPL_U8 dstBuffer[]={1,1,1,1, 0,0,0,1, 1,0,1,0, 1,0,1,1} |
| **CSPL_16BitIntsToBuffer** | This function will copy data from a short integer array (i.e. two byte integer) to an array of single byte values. |
| **CSPL_32BitIntsToBuffer** | This function is similar to *CSPL_16BitIntsToBuffer* with the difference that it operates on 4-byte integer array. |
| **CSPL_FloatsToBuffer** | This function is similar to *CSPL_16BitIntsToBuffer* with the difference that it operates on a floating point array. Each floating point value is represented by four bytes in the destination buffer. |
| **CSPL_StringToBuffer** | This function copies a 'C' string into an array of bytes including the terminating NULL character. It can be used send a character string from the user database to a Modbus master. |
| **CSPL_BufferTo16BitInts** | This function performs the reverse task as the *CSPL_16BitIntsToBuffer* function by mapping an array of bytes to a short integer array. |
| **CSPL_BufferTo32BitInts** | This function performs the reverse task as the *CSPL_32BitIntsToBuffer* function by mapping an array of bytes to a long integer (4-byte) array. |
| **CSPL_BufferToFloats** | This function performs the reverse task as the *CSPL_FloatsToBuffer* function by mapping an array of bytes to a floating point array |
| **CSPL_BufferToString** | This function performs the reverse task as the *CSPL_StringToBuffer* function by mapping an array of bytes to a 'C' string. |

# Configure Diagnostics

MSPL-C has embedded debugging code to printout out useful information to enable users to analyse, debug and diagnose the function of the library. Such code can be enabled only during initial development and disabled later to save code space as well as to decrease the CPU utilisation of the library.
The type of debugging statements output by the library also controlled at four levels as discussed in section 2.6.1
All diagnostics settings are done using 'C' macros making them configurable only at compile time and not at run time. So configuring diagnostics can be done with the following steps:

**Step-1: Select debugger level**

**Step-2: Include or exclude Formatted I/O support**

**Step-3: Implement the debug "sink"**

### 2.6.1 Step-1: Select debugger level

Enabling the debugger and setting the debug level is done by defining a value for the **DEBUG_LEVEL** macro. This macro is defined in *MSPL_UserIf.h*
e.g.

```
#define DEBUG_LEVEL DEBUG_ERROR
```

This macro can be assigned one of the following values:

| Macro Value | Description |
|---|---|
| **DEBUG_NONE** | This value disables the debugger. No debugging statements are output from the library. This is the value you will use once your application has been fully tested and ready to be released. |
| **DEBUG_ERROR** | This value causes the debugger to output statements when any error occurs in the library. In a well tested application there should be very few occurrences of "error debugger statements". In a way, it's a good idea to set the debugger to this level during the initial period after a release is done in order to capture errors that might occur post-release. An example of an error condition is when the library finds that the Modbus packet that has arrived is larger than the buffer size configured. In this case the library outputs this message: *"Error: MSPL_ReadMbPdu: Buffer too small to read PDU"* |
| **DEBUG_WARNING** | This value causes the debugger to output relevant messages when errors occur or when conditions occur that could potentially lead to errors. An example of a warning is when the library receives a Modbus request with the function code set to an unsupported value. In this case the library outputs this message: *"Warning: Unsupported function code, sending exception response"* |
| **DEBUG_INFORMATION** | This value causes the debugger to output routine information that indicates the overall status of the library and also shows the flow of execution, in addition to error and warning messages. This is the setting you will use in diagnosing any errors reported in the application. For instance when the library receives a Modbus read request for *Coils*, it outputs the following informational message: *"==> FC=0x01 (Read Coils) "* |
| **DEBUG_VERBOSE** | This value causes the debugger to output messages that can be used for deep debugging. An example of such a message is when the library outputs the value of each byte of the Modbus packet received by it as well as that of the response. This setting is useful in diagnosing difficult problems but at the same time generates an overwhelming amount of messages that can get you lost. |

Back to the top

### 2.6.2 Step-2: Include or exclude Formatted I/O support

If a function like s*printf* that implements formatted I/O is supported on the platform, the library can make use of it to create more meaningful debugging messages. For instance if a Modbus request with an unsupported function code is received, the debug message will be formatted to contain the unsupported function code to make it easier to debug the problem.
Support for formatted I/O can be configured by setting the macro **STDIO_SUPPORTED** to a value of '1' and providing the name of the function to be used in the macro **FORMATTED_STRING_PRINT**. Both these macros are defined in *MSPL_UserIf.h*.

```
#define STDIO_SUPPORTED 1 // Enable formatted I/O support
#define FORMATTED_STRING_PRINT sprintf
```

Back to the top

### 2.6.3 Step-3: Implement the debug "sink"

The debugging messages output by the library have to be finally output to a physical device like a display, a printer or a serial terminal etc. This output device is referred to as the *debug sink*. To provide the flexibility of choosing the debug sink to the user, the library outputs its messages to a function called *MSPL_DebugPrint*. This function is defined in *MSPL_UserIf.c* but is left unimplemented (i.e. an empty function). Users should implement this function and sink the debug message passed as an argument to an appropriate device.
The format of this function is as below:

```
void MSPL_DebugPrint( CSPL_U8 networkNo, CSPL_U8 eventType,
                      char * debugMessage)
```

Parameters:

i. **networkNo** (IN): The network who Modbus instance generated this debug print. This paramter enables redirecting of debug prints from different networks to different sinks so that they do not all get jumbled.

ii. **eventType** (IN): Indicates the debug level at which this debug print was made. Possible values are one of: DEBUG_VERBOSE, DEBUG_INFORMATION, DEBUG_WARNING and DEBUG_ERROR. A possible use of this information is to print debug messages of different levels in different colours.

iii. **debugMessage** (IN): A null-terminated 'C' string containing the debug message.

Shown below is a very simple implementation of this hook function that adds a time stamp to the debugger message and prints it to the standard output device.

```
void MSPL_DebugPrint(CSPL_U8 networkNo, CSPL_U8 eventType, char* debugMessage)
{
    /* Add a time stamp to the debugger message & print it to the
       standard output */
    SYSTEMTIME st;
    GetLocalTime(&st);
    printf("%d:%d:%d.%03d - %s", st.wHour, st.wMinute, st.wSecond,
           st.wMilliseconds, debugMessage);
}
```

Back to the top

# Optimise MSPL-C

Design constrains change from one platform to another. While someone is constrained for Data Memory (RAM) space, someone else is short of Code (Program) Memory (ROM/Flash) while yet another is short of both. In order to accommodate MSPL within the design constraints of most users, we have provided mechanisms to save RAM, ROM or both. The following sub sections describe the steps involved in using each of these techniques.

### 2.7.1 Set optimal buffer sizes

The library uses memory buffers to store incoming Modbus packets before decoding them and to store response packets before transmitting them. The sizes of these two buffers can be controlled by limiting the maximum number of Modbus data items (i.e. coils, registers etc.) that a master can request in one Modbus transaction. For instance if a Modbus Master sends a *read* request for 100 registers in one packet, the resulting response packet size will be greater than 200 bytes in comparison to a read request for just 10 registers. You can configure the library to entertain requests that can fit into a specific buffer size by defining the following macros:

| Macro Name | Location | Remarks |
|---|---|---|
| RX_BUFFER_SIZE | MSPL_Userlf.h | Limits the size of incoming packets. If the incoming request packet size cannot be accommodated in this buffer size, the library outputs an "Error" debugger message, discards the received packet and sends no response to the master. |
| TX_BUFFER_SIZE | MSPL_Userlf.h | Limits the size of outgoing packets. **Note:** No check is made by the library to verify if a Modbus request results in a response packet whose size is larger than this size. |

#### 2.7.1.1 Modbus Block Size Macros

Modbus block size is the number of data items that a master zor operate upon in one Modbus transaction. The size of a Modbus packet is limited to 256 bytes for Modbus RTU and 260 bytes for Modbus TCP. This in effect itself limits the number of items that can be operated upon in one transaction as below:

| Transaction | Max permissible block size |
|---|---|
| **Read Coils, Read Discrete Inputs** | 2000 coils and Discrete Inputs respectively |
| **Read Holding Registers, Read Input Registers** | 125 registers |
| **Write Multiple Coils** | 1968 coils |
| **Write Multiple Registers** | 123 registers |

However, in order to receive and service Modbus transactions that stretch up to the above max permissible limits, a device needs a transmit and a receive buffer of 256 bytes (260 in case of Modbus TCP). This may not be available or necessary in small devices employing low end microcontrollers. MSPL provides a way of using a lower buffer sizes and a set of macros which can be used to filter out Modbus transactions that exceed a set limit for block size. They are:

- RD_BLK_SIZE_BITINFO
- WR_BLK_SIZE_BITINFO
- RD_BLK_SIZE_REGINFO
- WR_BLK_SIZE_REGINFO

These macros must be set along with RX_BUFFER_SIZE and TX_BUFFER_SIZE to optimize the use of memory.

### 2.7.2 Enable only the function codes you require

The code size occupied by the library can be minimized by including only the Modbus functions required in your application and excluding others. For instance, if your device has only digital inputs, there is no use of including support for Modbus function *Read Holding Register*. The library provides a set of macros using which you can selectively include or exclude Modbus functions.

#### 2.7.2.1 How to use the Modbus function support macros

To include support for a Modbus function, set the macro to '1', else set it to '0'.
Example:

```
#define INCLUDE_READ_COILS 1 // Adds code to support Read Coils function
#define INCLUDE_READ_COILS 0 // Excludes code that implements Read Coils
                             // function
```

#### 2.7.2.2 How does the library respond to an unsupported function request

When the library receives a request for an unsupported Modbus function it responds with Modbus Exception code 0x01 (ILLEGAL FUNCTION).

#### 2.7.2.3 List of supported macros

| Macro Name | Modbus Function Affected |
|---|---|
| INCLUDE_READ_COILS | Read Coils (Function Code 0x01) |
| INCLUDE_READ_DISCRETE_IP | Read Discrete Inputs (Function Code 0x02) |
| INCLUDE_READ_INPUT_REGS | Read Input Registers (Function Code 0x04) |
| INCLUDE_READ_HOLDING_REGS | Read Holding Registers (Function Code 0x03) |
| INCLUDE_WRITE_COILS | Write Multiple Coils (Function Code 0x0F) |
| INCLUDE_WRITE_REGISTERS | Write Multiple registers (Function Code 0x10) |

### 2.7.3 Reduce Code Memory size by excluding support for message counters

The library maintains counters for total Modbus messages received by it, total messages responded by it, total errors encountered and so on. This functionality is optional, not mandatory, as per the Modbus standard. So you may exclude this functionality by setting macro INCLUDE_MSG_CTRS to zero.

```
#define INCLUDE_MSG_CTRS 0 // Exclude message counters related code
```

### 2.7.4 Reduce Code Memory size by configuring CRC macros (Modbus RTU only)

The amount of *Code Memory* (sometimes called *Program Memory*) used by the library can be reduced using two technics.

***Method 1:*** **Move CRC tables into Data Memory (RAM)**

**Steps**

a) Open file MSPL_Userlf.h
b) Locate the definition of macro *CRC_TABLE_LOCATION*
c) Change its value to *IN_RAM*. The modified line should look like this:

```
#define CRC_TABLE_LOCATION      IN_RAM
```

d) Rebuild your project. You should see a reduction in code size by approximately 512 bytes and a corresponding increase in RAM usage.

**Description**

Two tables of 256 constant values are used in computing CRC bytes. The location of these tables is configurable. The above steps cause the tables to be stored in data memory. This saves code memory at the expense of data memory by moving the tables into RAM. Since RAM is faster than ROM access, this method may also improve the efficiency of code execution.

***Method 2:*** **Eliminate CRC table storage by computing table contents dynamically**

**Steps**

a) Open file MSPL_Userlf.h
b) Locate the definition of macro *CRC_TABLE_LOCATION*
c) Change its value to *CREATE_DYNAMIC*. The modified line should look like this:

```
#define CRC_TABLE_LOCATION      CREATE_DYNAMIC
```

d) Rebuild your project. You should see a reduction in code size by approximately 512 bytes *without* a proportional increase in RAM usage.

**Description**

This setting eliminates the two CRC tables altogether by computing the values of this table dynamically as and when required. This saves both code memory as well as data memory. However, since the CRC table contents are computed twice (once for verifying the CRC of received request ADU and again for computing the CRC for the response ADU) for every Modbus transaction, this method considerable increases the load on the CPU. This may lead to slower response times from the library.

### 2.7.5 Reduce Data Memory (RAM) size by configuring CRC macros

The amount of *Data Memory* (sometimes called *as RAM*) used by the library can be reduced using two techniques.

***Method 1:*** **Move CRC tables into Code Memory (ROM)**

**Steps**

a) Open file MSPL_Userlf.h
b) Locate the definition of macro *CRC_TABLE_LOCATION*
c) Change its value to *IN_ROM*. The modified line should look like this:

```
#define CRC_TABLE_LOCATION      IN_ROM
```

d) Rebuild your project. You should see a reduction in RAM usage but an increase in the code size.

**Description**

Two tables of 256 constant values are used in computing CRC bytes. The location of these tables is configurable. The above steps cause the tables to be stored in code memory. This saves data memory (RAM) at the expense of code memory (ROM) by moving the tables into ROM.

***Method 2:*** **Eliminate CRC table storage by computing table contents dynamically**

**Steps**

a) Open file MSPL_Userlf.h
b) Locate the definition of macro *CRC_TABLE_LOCATION*
c) Change its value to *CREATE_DYNAMIC*. The modified line should look like this:

```
#define CRC_TABLE_LOCATION      CREATE_DYNAMIC
```

d) Rebuild your project. You should see a reduction in code size by approximately 512 bytes *without* a proportional increase in RAM usage.

## Build and test your port with the supplied Modbus Protocol Tester

If you have reached here then you have:

- Created a project in your IDE and added the MSPL files to it
- Implemented the hook functions to glue the library to your platform's physical interface
- Implemented the hook functions to glue the library to your application's database or are using the simulated database
- Configured the debugger
- Defined appropriate values for various open macros to set the Endian style, optimise the memory utilisation etc.

Use the facilities of the IDE now to compile and build the project and download it to your target. You can test this port with the bundled Modbus Protocol Tester application.

### 2.8.1 Modbus Protocol Tester - Overview

The Modbus Protocol Tester or MPT in short is a simple Windows Modbus Master application that enables a user to send Modbus requests to a slave device and decode the response. It can be configured to poll specified data points in the slave device at a periodic interval and generate a log if any error occurs. It can display raw Modbus packets as well as the data extracted from a Modbus packet. The data can be shown as raw Modbus data (bits and words) or as interpreted data (integers, floats and so on). It is an excellent light-weight test tool to validate a Modbus device's compliance to the supported function codes and also perform stress tests on it by bombarding it with Modbus requests.

### 2.8.2 Installing Modbus Protocol Tester

The setup file for the Modbus Protocol Tester is in the "..\MPT" folder of MSPL-C package. Run setup.exe from this folder and follow the onscreen instructions to complete the installation.

### 2.8.3 Help on using Modbus Protocol Tester

Modbus Protocol Tester online user's manual can be found at http://www.colwaysolutions.com/topic-based-user-manual.html

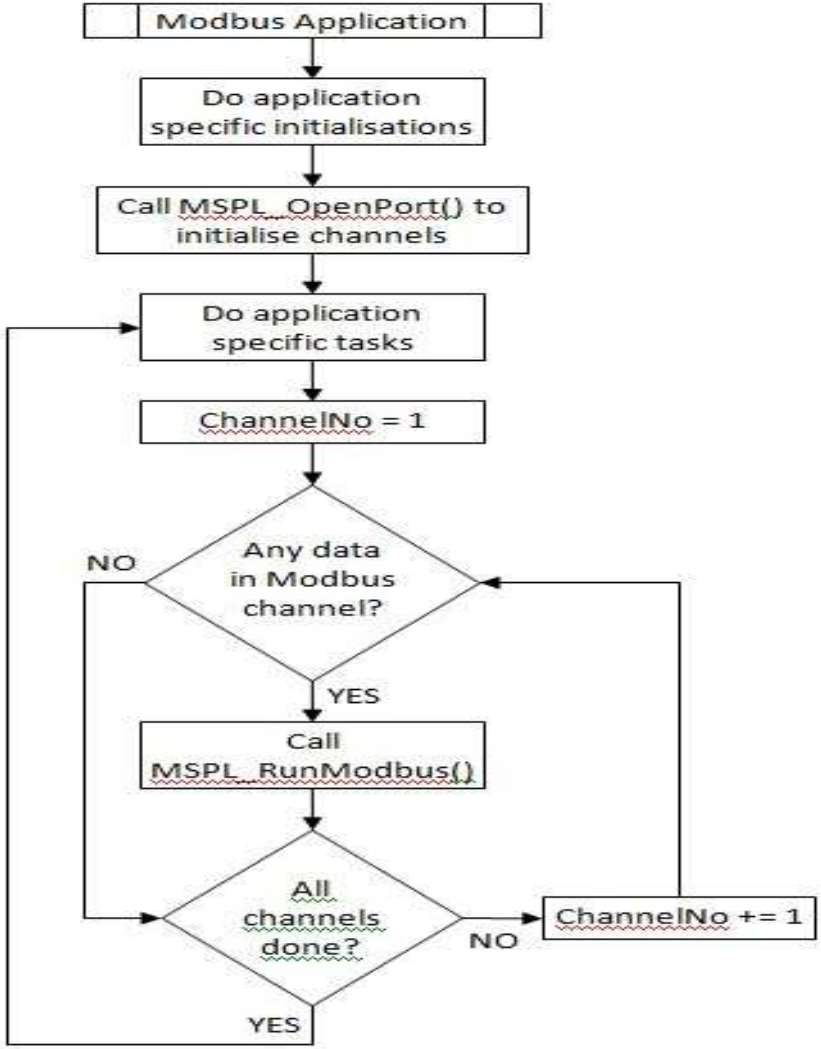
## Making Calls into MSPL-C APIs

Once you have ported the library to your platform, it is time to make calls into its API's. The following table shows a list of API's that may be called by the user's application:

| API | When to call | Mandatory? | Remarks |
|---|---|---|---|
| **MSPL_RunModbus** | Periodically for every channel, as soon as some bytes have arrived into the channel. | Yes | –> This is the main entry point into the library, also called as the trigger function.<br>–> This function triggers the stack which checks if data has arrived on the specified channel and processes it.<br>–> This function must be called after the communication channel's device driver is queried to see if some data has been received in its buffer. |
| **MSPL_OpenPort** | On program start up, once for every communication channel to be opened and initialised. | No (optional) | –> The user is free to perform channel initialisation outside of the library in which case this function need not be implemented and/or called. |
| **MSPL_ClosePort** | Once per channel when Modbus communication is no longer required on that channel. | No (optional) | –> In applications where Modbus communication is expected to be active until the device is switched OFF, this function need not be called at all.<br>–> As for *MSPL_OpenPort*, user may choose to implement channel de-initialisation code outside the library in which case this function need not be implemented or called. |
| **MSPL_GetMessageCounters** | When the value of statistical counters maintained by the stack are required. | No (optional) | –> Details of the counters can be found in the function reference section for *MSPL_GetMessageCounters()* |

## Flowchart for MSPL-C API Invocation

Diagram below shows a flowchart of invocation of the *MSPL_OpenPort* function and *MSPL_RunModbus* function.



## MSPL-C Configurator for Easy Configuration of the Library

The MSPL-C Configurator is a Windows software that you can use to graphically set values for macros of the library. This tool directly modifies the MSPL_UserIf.c file in the same manner as you would change values for macros manually.

## How to use the MSPL-C Configurator

The following is the procedure to use the MSPL-C Configurator utility:

**Step 1.** Click "Browse" to navigate to the folder containing the MSPL-C source files and select the file *MSPL_UserIf.c* (only this file is required to begin using the utility). On choosing the file, all fields in the UI will get populated with the values of macros read from this file. The pathname of the folder selected appears in the text box labelled "MSPL-C Location".

**Step 2.** Modify values of any of the configuration parameter that you desire.

**Step 3.** In the process if you intend to discard any changes you have done and reset the UI to the values in *MSPL_UserIf.c*, press "Reload"

**Step 4.** Once you are satisfied with the changes made by you save them back to the *MSPL_UserIf.c* file by pressing "Save Changes". This step will modify this file by changing the values of the macros contained in it. So ensure that the file is writable.

## MSPL-C Configurator Settings

Each control on the configurator window controls the value of a macro. Table below provides the full list of controls and the macros that they affect. Descriptions of the macros themselves are covered in various sub-sections of the section "How to port MSPL-C to your platform".

| Control Group | Control Name | Macro affected | Value set for macro |
|---|---|---|---|
| **CRC Table Location Settings** | CRC Tables in ROM | CRC_TABLE_LOCATION | IN_ROM |
| | CRC Tables in RAM | CRC_TABLE_LOCATION | IN_RAM |
| | Dynamic CRC Tables | CRC_TABLE_LOCATION | CREATE_DYNAMIC |
| **Enable Function Groups** | Read Coils | INCLUDE_READ_COILS | 1 if checked, else 0 |
| | Read Discrete Inputs | INCLUDE_READ_DISCRETE_IP | 1 if checked, else 0 |
| | Read Holding Registers | INCLUDE_READ_HOLDING_REGS | 1 if checked, else 0 |
| | Read Input Registers | INCLUDE_READ_INPUT_RaEGS | 1 if checked, else 0 |
| | Write Coils | INCLUDE_WRITE_COILS | 1 if checked, else 0 |
| | Write Holding Registers | INCLUDE_WRITE_REGISTERS | 1 if checked, else 0 |
| **Debugger Settings** | Enable Debugging | DEBUG_LEVEL | DEBUG_NONE if unchecked else one of the values below |
| | Verbose | DEBUG_LEVEL | DEBUG_VERBOSE |
| | Information | DEBUG_LEVEL | DEBUG_INFORMATION |
| | Warning | DEBUG_LEVEL | DEBUG_WARNING |
| | Error | DEBUG_LEVEL | DEBUG_ERROR |
| | STDIO Supported | STDIO_SUPPORTED | 1 if checked, else 0 |
| **Modbus Mode** | Modbus TCP | MODBUS_MODE | MODBUS_TCP |
| | Modbus RTU | MODBUS_MODE | MODBUS_RTU |
| **Endian Style** | Big Endian | ENDIAN_STYLE | BIG_ENDIAN |
| | Little Endian | ENDIAN_STYLE | LITTLE_ENDIAN |
| **Modbus Block Size Limits** | Read Bit Status | RD_BLK_SIZE_BITINFO | Value entered in text box. Permitted range: 8-2000. |
| | Read Register Value | RD_BLK_SIZE_REGINFO | Value entered in text box. Permitted range: 1-125. |
| | Write Bit Status | WR_BLK_SIZE_BITINFO | Value entered in text box. Permitted range: 8-1968. |
| | Write Register Value | WR_BLK_SIZE_REGINFO | Value entered in text box. Permitted range: 1-123. |
| **Variable Location Modifiers** | Const variables in ROM | CRC_TABLE_LOC_MODIFIER | Text entered in text box |
| | Variables in external RAM | DATA_IN_XRAM | Text entered in text box |
| **Network Settings** | No. of Networks | MAX_NETWORKS | Value entered in text box |

## Using MSPL-C in a multitasking Environment

Two most common types of multitasking environments in use are the *process based RTOS* and a *task based RTOS*. They differ in the following characteristics:

| Process based RTOS | Task based RTOS |
|---|---|
| Each process has a dedicated global memory space. | All tasks share a common global memory space. |
| A global variable with the same name can exist between two processes. | A global variable created in one file is visible to all tasks. So global variables should have unique names. |
| Functions of a library used by a process need not be *re-entrant* or *thread-safe* since a call to it by multiple processes will always act only on the global data of the calling process, if any. | Since function calls by any of the tasks act on shared global data, functions of a library used by multiple tasks simultaneously need be *re-entrant* or *thread-safe*. |
| Examples: WinCE, OS9 | Examples: VxWorks, uC/OS II, MQX, embOS |

MSPL-C has its set of global variables. So these differences have an impact on how MSPL-C can be used in these multitasking environments. Consider the following guidelines when using MSPL-C in an RTOS:

a. In a process based RTOS, fork multiple processes to support many Modbus TCP connections with each process handling one connection. However, set MAX_NETWORKS macro in MSPL_UserIf.h to one (1) only. Each process behaves as if it is the only Modbus device in the system.

b. In a task based RTOS, fork multiple tasks to support many Modbus TCP connections. In this case the MAX_NETWORKS macro should be set to the maximum number of tasks that will simultaneously use the library.

# MSPL - C Reference

## 6.1 MSPL-C Data Types

These data types are defined in *CSPL_MbDefs.h*

| MSPL-C Data type | Native definition |
|---|---|
| CSPL_U8 | `unsigned char` |
| CSPL_U16 | `unsigned short int` |
| CSPL_U32 | `unsigned int` |
| CSPL_I8 | `char` |
| CSPL_I16 | `short int` |
| CSPL_I32 | `int` |
| CSPL_BOOL | `typedef enum _CSPL_BOOL { CSPL_FALSE, CSPL_TRUE }CSPL_BOOL;` |

## 6.2 MSPL-C Function Reference

### 6.2.1 MSPL_UserInit

| Function name | MSPL_UserInit | |
|---|---|---|
| **Description** | This function is called by MSPL_Init() in order to allow any user/application specific initialisation to be done. This is a good place, for instance, to create/link to any synchronisation semaphores, initialise your own application globals or print some start up messages. | |
| **Returns** | **CSPL_BOOL**. A status code. This code is passed back by MSPL_Init() to its caller who can take appropriate action on failure. | |
| **Possible return values** | CSPL_TRUE - All user/application specific initialisation went OK.<br>CSPL_FALSE - Something went wrong during user de-initialisation. | |
| **Arguments** | None | |
| **Called by** | Library | |
| **User Implements?** | Yes | |

### 6.2.2 MSPL_UserDeInit

| Function name | MSPL_ UserDeInit | |
|---|---|---|
| **Description** | This function is called by the MSPL_DeInit() function of the library in order to allow any user/application specific de-initialisation/cleanup to be done. This is a good place, for instance, to close handles to any operating system objects that the application code might be using. | |
| **Returns** | **CSPL_BOOL**. A status code. This code is passed back by MSPL_DeInit() to its caller who can take appropriate action on failure. | |
| **Possible return values** | CSPL_TRUE - All user/application specific de-initialisation went OK.<br>CSPL_FALSE - Something went wrong during user de-initialisation. | |
| **Arguments** | None | |
| **Called by** | Library | |
| **User Implements?** | Yes | |

### 6.2.3 MSPL_OpenPort

| Function name | MSPL_ OpenPort | |
|---|---|---|
| **Description** | This function should open communication port and initialise it so as to get it ready for receiving Modbus packets and sending responses. This is the place to set all communication parameters like baud rate, parity, port timeouts etc. This function is not internally called by the library but must be called by the user during start up of his application, once for each port that will support Modbus communication. | |
| **Returns** | **CSPL_U8**. A value indicating if the specified port was opened and initialised successfully or not. | |
| **Possible return values** | CSPL_TRUE - The specified port was opened and initialised successfully.<br>CSPL_FALSE - The specified port could not be opened or initialised. | |
| **Arguments** | CSPL_U8 networkNo | A number identifying the "port" or channel used for Modbus communication that is to be initialised. |
| **Called by** | User application | |
| **User Implements?** | Yes | |

### 6.2.4 MSPL_ClosePort

| Function name | MSPL_ ClosePort | |
|---|---|---|
| **Description** | The implementation of this function should close the specified port and release all resources held by it. This function is not called directly by the library. It must instead be called by the user of the library when Modbus support on a communication port is no longer required. | |
| **Returns** | **CSPL_U8**. A value indicating success or failure of the function. | |
| **Possible return values** | CSPL_TRUE - The port was closed successfully.<br>CSPL_FALSE - The port could not be closed successfully. | |
| **Arguments** | CSPL_U8 networkNo | A number identifying the "port" or channel to be closed. |
| **Called by** | User application | |
| **User Implements?** | Yes | |

### 6.2.5 MSPL_CheckSlaveId

| Function name | MSPL_CheckSlaveId | |
|---|---|---|
| **Description** | This function is called by the library soon after it reads the Slave ID field from a Modbus ADU in order to know if the Modbus packet is intended for this device or not. The implementation of this function should return a value indicating if the packet is to be processed further or not. | |
| **Returns** | **CSPL_BOOL**. A value indicating if the slave ID passed identifies this device or not. | |
| **Possible return values** | CSPL_TRUE - Slave ID passed identifies this device and so this Modbus ADU may be processed further.<br>CSPL_FALSE - Slave ID passed does not identify this device. | |
| **Arguments** | CSPL_U8 slaveID | The slave ID to be validated. |
| | CSPL_U8 networkNo | The network on which this Modbus ADU was received. this parameter may be ignored if the device presents itself on all the networks with the same slave ID. |
| **Called by** | Library | |
| **User Implements?** | Yes | |

### 6.2.6 MSPL_ValidateAddresses

| Function name | MSPL_ ValidateAddresses | |
|---|---|---|
| **Description** | This function is called by the library to check if the combination of data address range and the function code in the Modbus PDU is valid for this device. | |
| **Returns** | **CSPL_BOOL**. A value indicating if the address range is valid or not. If not, the library responds to this ADU with an ILLEGAL DATA ADDRESS (0x02) Exception response. | |
| **Possible return values** | CSPL_TRUE - The address range is valid.<br>CSPL_FALSE - The address range is not valid. | |
| **Arguments** | CSPL_U8 slaveID | A single byte value containing the slave ID of the ADU for which the data addresses are to be validated.<br>The slave ID is useful in cases where the library is being used to implement a virtual multi slave device where validation of addresses will be different for different slave IDâ€™s. If the device behaviour is independent of the slave ID to which a Modbus request is addressed to, then this parameter may be ignored. |
| | CSPL_U8 networkNo | The network on which this Modbus ADU was received. If the device behaviour is independent of the network on which a Modbus request is received, then this parameter may be ignored. |
| | CSPL_U8 functionCode | A single byte value of the Modbus function code associated with the data addresses to be validated. |
| | CSPL_U16 startAddress | A two-byte value that is the first address in the range to be validated. |
| | CSPL_U16 noOfItems | A two-byte value that is the number of addresses starting from *startAddress* that are to be validated. |
| **Called by** | Library | |
| **User Implements?** | Yes | |

### 6.2.7 MSPL_ReadUserData

| Function name | MSPL_ ReadUserData | |
|---|---|---|
| **Description** | The library calls this function for all PDU's that contain a Modbus "read" service request and expects its implementation to copy the relevant data from the user's database to the buffer it passes as one of the arguments. The buffer passed for holding the data must be filled in a specific format as indicated below, the format itself being data type specific. Helper functions are provided in file CSPL_Utils.c to assist the user in easily filling this buffer in the correct format. The formats have been chosen to keep the usage of memory to the minimum.<br>**Type of data** / **Format in which data is to be stored in pBuffer**<br>Bit Status: Bit stuffed as 8 status information in one byte<br>Registers: One register as two consecutive bytes in big endian format | |
| **Returns** | **CSPL_U8**. A value indicating if the read request was processed successfully or not. | |
| **Possible return values** | = 0 - if the service request executed successfully<br>> 0 - if the service request failed in which case the library sends a SLAVE DEVICE FAILURE Exception response (code 0x04) to the Modbus master. | |
| **Arguments** | CSPL_U8 slaveID | A single byte value containing the slave ID of the ADU for which the data addresses are to be validated.<br>The slave ID is useful in cases where the library is being used to implement a virtual multi slave device where validation of addresses will be different for different slave ID's. If the device behaviour is independent of the slave ID to which a Modbus request is addressed to, then this parameter may be ignored. |
| | CSPL_U8 networkNo | The network on which this Modbus ADU was received. If the device behaviour is independent of the network on which a Modbus request is received, then this parameter may be ignored. |
| | CSPL_U8 functionCode | A single byte value of the Modbus function code that defines the Modbus service request. This parameter can be checked to see what kind of data (coil, discrete input, register etc.) is being requested for. |
| | CSPL_U16 startAddress | A two-byte value that is the first address in the range of data being requested for. |
| | CSPL_U16 noOfItems | A two-byte value that is the number of data items starting from startAddress that are being requested for. |
| | CSPL_U8 *pBuffer | Pointer to an array of bytes into which the requested data must be copied into in the correct format.<br>Appropriate helper functions may be used in stuffing the data in the right format into this buffer. |
| **Called by** | Library | |
| **User Implements?** | Yes | |

### 6.2.8 MSPL_WriteUserData

| Function name | MSPL_ WriteUserData | |
|---|---|---|
| **Description** | This function is used transfer data from a Modbus PDU to the userâ€™s database. The library calls this function for all PDU's that contain a Modbus "write" service request and expects its implementation to copy the relevant data from the buffer it passes as one of the arguments to the userâ€™s database. The buffer passed contains the PDU data in a specific format as described below. Helper functions are provided in file CSPL_Utils.c to assist the user in easily decoding and copying data in this buffer to his application's database. The formats have been chosen to keep the usage of memory to the minimum.<br>**Type of data** / **Format in which data is stored in pBuffer**<br>Bit Status: Bit stuffed as 8 status information in one byte<br>Registers: One register as two consecutive bytes in big endian format | |
| **Returns** | **CSPL_U8**. A value indicating if the write request was processed successfully or not. | |
| **Possible return values** | = 0 - if the service request executed successfully<br>> 0 - if the service request failed in which case the library sends a SLAVE DEVICE FAILURE Exception response (code 0x04) to the Modbus master. | |
| **Arguments** | CSPL_U8 slaveID | A single byte value containing the slave ID of the ADU for which the data addresses are to be validated.<br>The slave ID is useful in cases where the library is being used to implement a virtual multi slave device where validation of addresses will be different for different slave ID's. If the device behaviour is independent of the slave ID to which a Modbus request is addressed to, then this parameter may be ignored. |
| | CSPL_U8 networkNo | The network on which this Modbus ADU was received. If the device behaviour is independent of the network on which a Modbus request is received, then this parameter may be ignored. |
| | CSPL_U8 functionCode | A single byte value of the Modbus function code that defines the Modbus service request. This parameter can be checked to see what kind of data (coil, discrete input, register etc.) is present in the buffer and which Modbus service is being used to write the data. |
| | CSPL_U16 startAddress | A two-byte value that is the first address in the range of data being written to. |
| | CSPL_U16 noOfItems | A two-byte value indicating the number of data items starting from startAddress that are being written to.<br>If *functionCode* is 0x05 or 0x06 then *noOfItems* will be '1'. |
| | CSPL_U8 *pBuffer | Pointer to an array of bytes containing the data. The library fills this buffer with data from the received PDU. |
| **Called by** | Library | |
| **User Implements?** | Yes | |

### 6.2.9 MSPL_ReadPort

| Function name | MSPL_ ReadPort | |
|---|---|---|
| **Description** | This function is called by the library to read a Modbus packet from a communication port. | |
| **Returns** | **CSPL_U8**. A value indicating success or failure of the function. | |
| **Possible return values** | CSPL_TRUE - if the function is able to read at least one byte from the port before a read timeout occurs. The actual number of bytes read should be stored in *pNoOfBytesRead*.<br>CSPL_FALSE - if the function is unable to read any byte from the port before a read timeout occurs or if it encounters an error in reading the port. In this case an error code indicating the reason for failure should be stored in *pErrorCode* argument. | |
| **Arguments** | CSPL_U8 networkNo | A number identifying the "port" to be read. |
| | CSPL_U16 noOfBytesToRead | The number of bytes to read on this port. |
| | CSPL_U16 *pNoOfBytesRead | A pointer to the variable that receives the actual number of bytes read. |
| | CSPL_U8 *pBuffer | A pointer to the buffer that receives the data read from the port. |
| | CSPL_U8 *pErrorCode | A pointer to the variable that receives an error code in case of failure of this function. |
| **Called by** | Library | |
| **User Implements?** | Yes | |

### 6.2.10 MSPL_WritePort

| Function name | MSPL_ WritePort | |
|---|---|---|
| **Description** | This function is called by the library to write a Modbus response packet to a communication port. | |
| **Returns** | **CSPL_U8**. A value indicating success or failure of the function. | |
| **Possible return values** | CSPL_TRUE - if the function is able to write at least one byte to the port before a write timeout occurs. The actual number of bytes written should be stored in *pNoOfBytesWritten*.<br>CSPL_FALSE - if the function is unable to write any byte to the port before a write timeout occurs or if it encounters an error in writing to the port. In this case an error code indicating the reason for failure should be stored in *pErrorCode* argument. | |
| **Arguments** | CSPL_U8 networkNo | A number identifying the "port" to be read. |
| | CSPL_U16 noOfBytesToWrite | The number of bytes to write to this port. |
| | CSPL_U16 * pNoOfBytesWritten | A pointer to the variable that receives the actual number of bytes written. |
| | CSPL_U8 *pBuffer | A pointer to the buffer containing the data to be written to the port. |
| | CSPL_U8 *pErrorCode | A pointer to the variable that receives an error code in case of failure of this function. |
| **Called by** | Library | |
| **User Implements?** | Yes | |

### 6.2.11 MSPL_DebugPrint

| Function name | MSPL_ DebugPrint | |
|---|---|---|
| **Description** | The library calls this function to output a debug message. Users may implement this function to sink the debug message to an output device of their choice (e.g. to a printer, to an LCD, to a file and so on.). This function is called only when debugging is enabled by way of macro DEBUG_LEVEL. | |
| **Returns** | None | |
| **Arguments** | CSPL_U8 networkNo | The network on which a Modbus transaction was occurring that caused this debug message. This argument enables redirecting of debug prints from different networks to different sinks so that they do not all get jumbled. |
| | CSPL_U8 eventType | The debug level of this message. Possible values are:<br>DEBUG_VERBOSE<br>DEBUG_INFORMATION<br>DEBUG_WARNING<br>DEBUG_ERROR.<br>A possible use of this information is to print debug messages of different levels in different colours. |
| | CSPL_CHAR* debugMessage | A null-terminated 'C' string containing the debug message. |
| **Called by** | Library | |
| **User Implements?** | Yes | |

### 6.2.12 MSPL_RunModbus

| Function name | MSPL_RunModbus | |
|---|---|---|
| **Description** | This is the entry point function into the library which must be run periodically to execute the Modbus stack. When called, it reads the channel (passed as an argument) to check if any data has been received. If so, it attempts to process the packet received as a Modbus frame and if required sends out a response. | |
| **Returns** | **CSPL_U8**. A status code as shown in section below titled "**Status codes returned by function *MSPL_RunModbus***" | |
| **Arguments** | CSPL_U8 networkNo | The channel on which this function must look for a Modbus packet. The library passes this parameter to every hook function that it calls from MSPL_UserIf.h. Since this function processes one channel at a time, it must be called once for every channel configured for Modbus in your system. |
| **Called by** | User application | |
| **User Implements?** | No | |

### *6.2.13* Status codes returned by function *MSPL_RunModbus*

The following error codes may be returned by the main entry point function MSPL_RunModbus. They are defined in *CSPL_MbDefs.h*

| Error | Code | Remarks |
|---|---|---|
| MSPL_NO_ERROR | 0x00 | No error was encountered and the function executed successfully |
| UNKNOWN_ERROR | 0x01 | An unknown error occurred reading / writing to port. This indicates that the underlying device driver API for read/write returned an unknown code when invoked. |
| INVALID_HANDLE | 0x02 | An invalid handle or path ID was used to read from / write to the port. |
| INVALID_NETWORKNUM | 0x03 | An uninitialized network number was passed as a parameter. Indicates that an attempt was made to use a channel that has not been initialised with a call to MSPL_OpenPort(). |
| READ_WRITE_FAIL | 0x04 | Device failure reading / writing to port. Indicates that the underlying device driver API for read/write returned an error code. |
| READ_WRITE_TIMEOUT | 0x05 | Timeout occurred reading / writing bytes. Indicates that the library called *MSPL_ReadPort* which returned with no data but a timeout. |
| ID_MISMATCH | 0x06 | The slave ID found in the Modbus request does not match this device. Indicates that the library encountered a message that was directed to a different slave. In case of Modbus RTU, this could occur frequently when using a shared bus like RS485 whereas in case of Modbus TCP, this error code indicates a true errpr. |
| CRC_ERR | 0x07 | The message contained incorrect CRC Bytes. Indicates a corrupt message. Modbus RTU only. |
| BUFFER_TOO_SMALL | 0x08 | The request message has more bytes than the available size of buffer. Indicates that the master is trying to read or write too many Modbus data units that the block sizes configured for the library. |
| PORT_CLOSED | 0x09 | The communication port was closed when trying to read or write on it. This error commonly occurs when a TCP connection is closed just when the library was trying to read from the channel. |
| INVALID_FC | 0x0A | An invalid/unsupported function code was requested to be serviced. |

### 6.2.14 MSPL_GetMessageCounters

| Function name | MSPL_GetMessageCounters | |
|---|---|---|
| **Description** | This function provides the value of various message counters maintained by the stack. | |
| **Returns** | **MSPL_MB_MSG_CTRS**. A structure containing the counters maintained by the stack. See section below titled "Structure MSPL_MB_MSG_CTRS" for details of these counters. | |
| **Arguments** | CSPL_U8 networkNo | The channel whose counters are being requested. The stack maintains an exclusive set of counters for each channel. |
| **Called by** | User application | |
| **User Implements?** | No | |

#### 6.2.14.1 Structure MSPL_MB_MSG_CTRS

| Member | Data Type | Description of the counter |
|---|---|---|
| **busMsgCnt** | CSPL_U32 | Quantity of messages that the remote device has detected on the communication channel since its last restart, clear counters operation or power-up. |
| **busCommErrCnt** | CSPL_U32 | Quantity of CRC errors encountered by the remote device on this channel since its last restart, clear counters operation or power-up. This member is present only in Modbus RTU mode. |
| **busExcpErrCnt** | CSPL_U32 | Quantity of MODBUS exception responses returned by the remote device since its last restart, clear counters operation or power-up. |
| **slvMsgCnt** | CSPL_U32 | Quantity of messages addressed to the remote device, or broadcast on this channel, that the remote device has processed since its last restart, clear counters operation, or power-up. |

## Macro Reference

### 6.3.1 MODBUS_MODE

| Macro name | MODBUS_MODE | |
|---|---|---|
| **Description** | Controls the Modbus framing type followed by the library. | |
| **Permitted Values** | MODBUS_TCP | Sets framing type to Modbus TCP |
| | MODBUS_RTU | Sets framing type to Modbus RTU |
| **Remarks** | Two framing types are supported by the library - Modbus RTU and Modbus TCP. | |

### 6.3.2 ENDIAN_STYLE

| Macro name | ENDIAN_STYLE | |
|---|---|---|
| **Description** | Defines the Endian style of the processor running the library. | |
| **Permitted Values** | LITTLE_ENDIAN | Processor is of type Little Endian |
| | BIG_ENDIAN | Processor is of type Big Endian |
| **Remarks** | Since Modbus is Big Endian, appropriate conversion logic is required when the library is run on a Little Endian processor. The library uses this macro to run such conversion logic conditionally wherever required. | |

### 6.3.3 Macros for exclusion of unsupported Modbus functions

| Macro name | INCLUDE_READ_COILS<br>INCLUDE_READ_DISCRETE_IP<br>INCLUDE_READ_INPUT_REGS<br>INCLUDE_READ_HOLDING_REGS<br>INCLUDE_WRITE_COILS<br>INCLUDE_WRITE_REGISTERS | |
|---|---|---|
| **Description** | Selectively includes or excludes support for a Modbus function. | |
| **Permitted Values** | 1 | *Includes* source code for the related Modbus functions |
| | 0 | *Excludes* source code for the related Modbus functions |
| **Remarks** | ı Used to optimise library by excluding source code of unsupported functions thus making the library smaller.<br>ıı The macro names are self suggestive of the Modbus functions that they affect. | |

### 6.3.4 INCLUDE_MSG_CTRS

| Macro name | INCLUDE_MSG_CTRS | |
|---|---|---|
| **Description** | Controls inclusion or exclusion of code related to maintenance of message counters in the library. | |
| **Permitted Values** | 1 | *Includes* source code for supporting message counters |
| | 0 | *Excludes* source code for supporting message counters |
| **Remarks** | Used to compress the code size by excluding support for message counters. | |

### 6.3.5 RD_BLK_SIZE_BITINFO

| Macro name | RD_BLK_SIZE_BITINFO | |
|---|---|---|
| **Description** | Fixes the maximum limit to the number of *bit status information (both coils and discrete inputs)* that may be *requested* by a Master in one Modbus transaction. | |
| **Permitted Values** | 8 to 2000 | |
| **Remarks** | ı A smaller block size limit will enable setting a smaller value for macro TX_BUFFER_SIZE thus reducing the memory used by the library for holding Modbus response packets.<br>ıı If a Read Coils or Read Discrete Inputs request is received with the number of items set to more than RD_BLK_SIZE_BITINFO, the library responds with an ILLEGAL DATA ADDRESS (0x03) exception code. | |

### 6.3.6 RD_BLK_SIZE_REGINFO

| Macro name | RD_BLK_SIZE_REGINFO | |
|---|---|---|
| **Description** | Fixes the maximum limit to the number of *register values (both holding registers and input registers)* that may be *requested* by a Master in one Modbus transaction. | |
| **Permitted Values** | 1 to 125 | |
| **Remarks** | ı A smaller block size limit will enable setting a smaller value for macro TX_BUFFER_SIZE thus reducing the memory used by the library for holding Modbus response packets.<br>ıı If a Read Holding Registers or Read Input Registers request is received with the number of items set to more than RD_BLK_SIZE_REGINFO, the library responds with an ILLEGAL DATA ADDRESS (0x03) exception code. | |

### 6.3.7 WR_BLK_SIZE_BITINFO

| Macro name | WR_BLK_SIZE_BITINFO | |
|---|---|---|
| **Description** | Fixes the maximum limit to the number of *coils* that may be *written* to by a Master in one Modbus transaction. | |
| **Permitted Values** | 8 to 1968 | |
| **Remarks** | ı A smaller block size limit will enable setting a smaller value for macro TX_BUFFER_SIZE thus reducing the memory used by the library for holding Modbus request packets.<br>ıı If a Write Multiple Coils request is received with the number of items set to more than WR_BLK_SIZE_BITINFO, the library responds with an ILLEGAL DATA ADDRESS (0x03) exception code. | |

### 6.3.8 WR_BLK_SIZE_REGINFO

| Macro name | WR_BLK_SIZE_REGINFO | |
|---|---|---|
| **Description** | Fixes the maximum limit to the number of *holding registers* that may be *written* to by a Master in one Modbus transaction. A smaller block size limit will enable setting a smaller value for macro RX_BUFFER_SIZE thus reducing the memory used by the library for holding Modbus request packets. | |
| **Permitted Values** | 1 to 123 | |
| **Remarks** | ı A smaller block size limit will enable setting a smaller value for macro TX_BUFFER_SIZE thus reducing the memory used by the library for holding Modbus request packets.<br>ıı If a Write Multiple Registers request is received with the number of items set to more than WR_BLK_SIZE_REGINFO, the library responds with an ILLEGAL DATA ADDRESS (0x03) exception code. | |

### 6.3.9 RX_BUFFER_SIZE and TX_BUFFER_SIZE

| Macro name | RX_BUFFER_SIZE<br>TX_BUFFER_SIZE | |
|---|---|---|
| **Description** | These macros control the sizes of receive and transmit buffers that the library allocates for receiving Modbus requests and sending responses. | |
| **Permitted Values** | 7 - 256 | Modbus RTU |
| | 11 - 260 | Modbus TCP |
| **Remarks** | ı RX_BUFFER_SIZE must be set large enough for the library to support the block size limits specified by WR_BLK_SIZE_BITINFO and WR_BLK_SIZE_REGINFO.<br>ıı TX_BUFFER_SIZE must be set large enough for the library to support the block size limits specified by RD_BLK_SIZE_BITINFO and RD_BLK_SIZE_REGINFO.<br>ııı The recommended way to set these macros is by using the MSPL configurator which automatically calculates the values for the buffers based on the values set for the block size limiting macros.<br>ıv If these macros are manually set, ensure that the buffers are large enough to hold the MBAP header (Modbus TCP only) or the Slave/Server Address (Modbus RTU only), the Modbus PDU and the CRC bytes (Modbus RTU only) | |

### 6.3.10 STDIO_SUPPORTED

| Macro name | STDIO_SUPPORTED | |
|---|---|---|
| **Description** | Indicates to the library if the platform supports formatted I/O or not. | |
| **Permitted Values** | 1 | Formatted I/O supported |
| | 0 | Formatted I/O not supported |
| **Remarks** | ı This macro is used by the library when creating debug messages.<br>ıı If the value for this macro is 1, the library formats debug messages with relevant numerical information by using *sprint* formatted I/O function. If not, the debug messages are plain textual information only.<br>ııı If this macro is set to 1, then the macro *FORMATTED_STRING_PRINT* must also be set to the name of the function on your platform that provides standard 'C' *sprint* function-like functionality. Relevant header file may have to be included within *MSPL_UserIf.h* file to support your formatted I/O function. | |

### 6.3.11 FORMATTED_STRING_PRINT

| Macro name | FORMATTED_STRING_PRINT |
|---|---|
| **Description** | This macro must be set to the name of the 'C' function that can do a formatted print into a 'C' string. Usually the name of such a function is itself *sprint*. |
| **Permitted Values** | Name of formatted string print function. |
| **Remarks** | ı This macro is used only when STDIO_SUPPORTED is set to 1.<br>ıı See remarks under STDIO_SUPPORTED for more information. |

### 6.3.12 DEBUG_LEVEL

| Macro name | DEBUG_LEVEL | |
|---|---|---|
| **Description** | This macro is used to enable or disable output of debugging messages by the library and to set the type of instances for which a debug message is generated. | |
| **Permitted Values** | DEBUG_NONE | Debug message generation is disabled. |
| | DEBUG_ERROR | Debug messages are generated only when error conditions occur in the library execution. |
| | DEBUG_WARNING | Debug messages are generated only when error conditions or such other conditions occur in the library execution that could lead to potential error conditions. |
| | DEBUG_INFORMATION | In addition to generating debug messages under error and warning conditions messages are generated that provide a general status indication of the execution of the stack. |
| | DEBUG_VERBOSE | This setting is a superset of the above three settings. In addition to debug messages for all the above conditions, extensive messages are printed out with as much information for the user as would be required for deep debugging. |
| **Remarks** | ı As the debug level increases from DEBUG_NONE to DEBUG_VERBOSE, the code memory occupied by the library as well as the CPU utilisation by it increase.<br>ıı It is recommended to set the level to DEBUG_ERROR in the release version of your product. This will help catch errors in the field. | |

### 6.3.13 DEBUG_COLSIZE

| Macro name | DEBUG_COLSIZE |
|---|---|
| **Description** | When DEBUG_LEVEL is set to DEBUG_VERBOSE, the length of debug messages could become too large. This macro can be used to limit the no. of characters per debug message. |
| **Permitted Values** | 32 to 132 |
| **Remarks** | ı This macro is useful in ensuring that the debug messages *fit* the column size of your output device (e.g. some printers have 64 character column width while larger ones have 132 characters) |

### 6.3.14 CRC_TABLE_LOCATION

| Macro name | CRC_TABLE_LOCATION | |
|---|---|---|
| **Description** | This macro is used to control the manner in which the CRC tables are created and stored in the library thereby optimising the use of code and data memory. | |
| **Permitted Values** | CREATE_DYNAMIC | CRC tables are created dynamically during runtime and not pre-created and stored in RAM or ROM. |
| | IN_RAM | CRC tables are created once at the start of the program and stored in data memory (RAM). |
| | IN_ROM | CRC tables are stored in code memory (ROM) as a *const* array. |
| **Remarks** | ı This macro is used only in MODBUS RTU mode.<br>ıı The CREATE_DYNAMIC setting saves both data and code memory at the cost of lower performance during runtime since the CRC tables have to be created for every Modbus packet received.<br>ııı The IN_RAM setting saves ROM (code memory) at the cost of using more data memory. Access to the CRC tables could be faster since RAM access is faster than ROM access.<br>ıv The IN_ROM setting saves RAM (data memory) at the cost of using more code memory. | |

### 6.3.15 CRC_TABLE_LOC_MODIFIER

| Macro name | CRC_TABLE_LOC_MODIFIER |
|---|---|
| **Description** | This macro is used to set the keyword that will cause constant variables in code memory (ROM). |
| **Permitted Values** | The keyword used for forcing constant variables into code memory. E.g. the keyword '*const*' |
| **Remarks** | ı Many compilers by default may store constant variables in code memory. If so, set this macro to a blank. |

### 6.3.16 DATA_IN_XRAM

| Macro name | DATA_IN_XRAM |
|---|---|
| **Description** | This macro is used to set the keyword that will cause variables to be placed in external memory. |
| **Permitted Values** | The keyword used for forcing constant variables into code memory. E.g. the keyword '*xdata*' |
| **Remarks** | ı Microcontrollers have internal RAM (sometimes in the form of on-chip registers) and external RAM (also on-chip but not a part of the MCU core). This keyword is used to force program variables to be placed in the external RAM.<br>ıı The location of program variables also depends on the memory model of setting of the compiler. For instance a *large* memory model could by default place all program variables in external ram in which case this macro setting becomes irrelevant.<br>ııı The library uses this macro to modify the location of transmit buffer and the receive buffer since they form the major component of memory usage by the library. All other variables used in the library are placed in the default memory type defined by the memory model setting. |

### 6.3.17 MAX_NETWORKS

| Macro name | MAX_NETWORKS |
|---|---|
| **Description** | This macro must be set to the maximum number of communication channels that the library will communicate on. |
| **Permitted Values** | 1 to a reasonable maximum value. |
| **Remarks** | ı The library allocates as many sets of global variables as the value of this macro.<br>ıı The actual number of channels used for communication may be less than this value. |