User Manual - Modbus Master Protocol Library - ‘C’ Source Code

Revision 0.1
March 2018

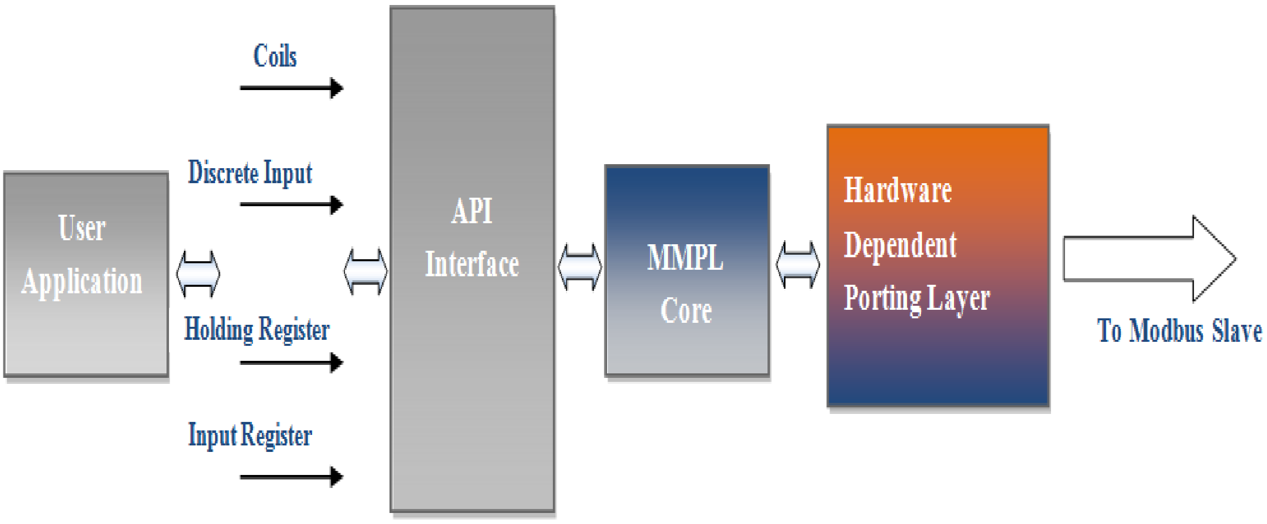
Table of Contents

1 Architecture

- 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

1.1 MMPL Block Schematic

Modbus Master Library: Components, Organization and Interconnections

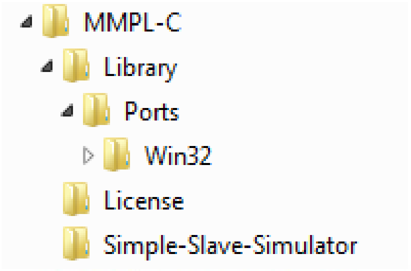


Components of Library:

S.No.	Module	Functionality	File Name
1	MMPL Core	Frame Parsing Packet Generation Deploy Modbus Functions Multi-level debugger	MMPL_C.c
2	Porting Layer	Links source code library to physical device and user application.	MMPL_UserIf.c *This file modified by user*

1.2 Directory Structure

Folders within MMPL-C package:-



- MMPL-C
  - Library
    - Ports
      - Win32
  - License
  - Simple-Slave-Simulator

Folder Name	Contents	Remarks
Library	Source files of MMPL-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 RTips Technologies regarding this library.
Simple Slave Simulator	Contains MSPL.exe	You will use this utility to listen to Modbus requests.
Ports	Ports of MMPL-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.

1.3 Files

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

File Type	Filename	Contents	Engineer Modifies ?
'C' Source	MMPL_C.c	Modbus communication protocol stack	No
	Main.c	Defines the entry point for the application.	Yes
	MMPL_UserIf.c	Platform dependent functions implemented by user "stubs" to receive platform dependent code. Refer to Win32 port for example.	Yes.
'C' header	MMPL_C.h	Header file for MMPL_C.c	No
	MMPL_Defs.h	Colway Solutions type and symbol definitions for maximum portability. CSPL_U16 CSPL_U132 etc.	Yes. Review and change for specific target
	MMPL_UserIf.h	Default values for all parameters. Refer to Win32 port for example	Yes. Extensive modification to complete port

Add MMPL 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 to explicitly configure all the above source files to be included in the build process.

1.4 Hooks and Macros

Hooks:

- The porting of MMPL-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

2 Porting

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

Step 1. Add MMPL-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 MMPL-C to the physical interface of your platform

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

Step 6. Configure diagnostics

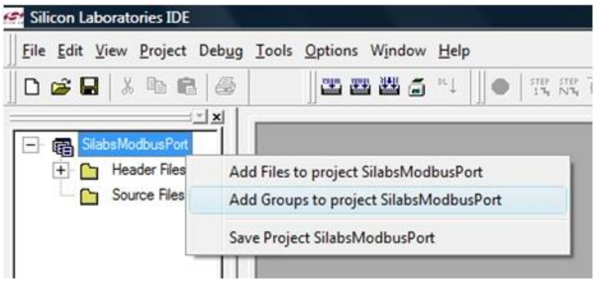
Step 7. Optimize MMPL-C

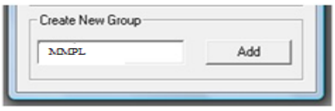
Step 8. Build and test your port with the supplied Simulator

2.1 Add a source code to your project

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

i.

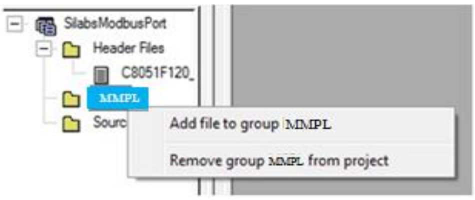

Silicon Laboratories IDE
File Edit View Project Debug Tools Options Window Help
SilabsModbusPort
Header Files
Source Files
Add Files to project SilabsModbusPort
Add Groups to project SilabsModbusPort
Save Project SilabsModbusPort

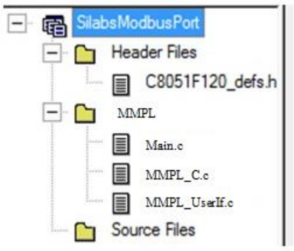


Create a new group called "MMPL" by right-clicking on the project name and choosing "Add Groups to project <proj name>" as shown below. Note that this step is optional.

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

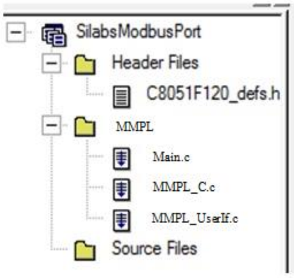
iii.





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

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



2.2 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 MMPL_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. MMPL_ShortIntsToBuffer) are "Endian-aware" - they are programmed check and ensure that transfers from interpreted data types of 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 are 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.

2.2.1 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 a 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.

2.3 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 MMPL_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.

2.4 Glue MMPL 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	MMPL_OpenPort	**i.** Use this function to open and configure communication channel **ii.** User application must call this function once for every channel supported by the device **iii.** A unique channel identification number is passed as an argument to this function. **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	MMPL_ReadPort	**i.** Library calls this function to read data from communication channel **ii.** Function typically calls device driver's "Read" API **iii.** A unique channel number is passed as an argument to identify the channel. **iv.** Caution: Blocking calls to device driver API's in this function will block execution of MMPL-C as well as the application code that is calling the library.
3.	Write to channel	MMPL_WritePort	**i.** Library calls this function to transmit data on communication channel **ii.** Function typically calls device driver's "Write" API **iii.** A unique channel number is passed as an argument to identify the channel. **iv.** Caution: Blocking calls to device driver API's in this function will block execution of MMPL-C as well as the application code that is calling the library.
4.	Close Port	MMPL_ClosePort	**i.** Use this function to close communication channel **ii.** User application calls this function when no Modbus communication is required **iii.** A unique channel number is passed as an argument to identify the channel.

2.5 Glue MMPL-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.

One function in the Main.c file facilitate the interface between the library and your application and database.

- DoModbusTransaction: Called by MMPL-C that drives Modbus communication on a network.

This function present a well defined interface that is fully documented in this manual.

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 Main.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)

The interface functions in the Main.c file operate upon this simulated database.

After testing with this database, you may replace it with your own. Modify the interface functions to operate on your database.

2.5.1 Using the Data Formatter to map ‘C’ data types to Modbus

MMPL-C provides you an extension to the Modbus specifications by supplying a set of functions in file *MMPL_C.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 *DecodeResponse* 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 *ConstrucRequest* toprovide the library with user data in a Modbus compliant format.

Following is a brief description of each function:

Function name	Description
MMPL_PackBits	This method bit-packs the destination buffer with bit status information from the source buffer. The source buffer is expected to contain bit status (i.e a value of 0 or 1) information in one byte per bit. This data is bit-packed as 8-bits per byte in the destination buffer.
MMPL_UnPackBits	This method unpacks the bits from the source buffer (which has data bit-packed as 8-bits per byte) and puts the bit status information (i.e a value of 0 or 1) into the destination buffer (in one byte per bit).
MMPL_ShortIntsToBuffer	This method puts data into the destination buffer in such a way that a pair of bytes of the destination buffer is used to hold the value of one two-byte register (source buffer).

MMPL_BufferToShortInts	This method puts data into the destination buffer in such a way that a pair of bytes of the source buffer is used to form the value of one two-byte register.

2.6 Configure diagnostics

MMPL-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 MMPL_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.
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.

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

If a function like sprintf 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'. This macro is defined in MMPL_UserIf.h.

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

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 MMPL_DebugPrint. This function is defined in MMPL_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 MMPL_DebugPrint( * debugMessage )
```

Parameters:

i. 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 MMPL_DebugPrint( 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);

}
```

2.7 Optimise MMPL-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 MMPL 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	MMPL_C.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	MMPL_C.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 Include only the function 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.

2.7.2.1 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).

2.7.3 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: Move CRC tables into Data Memory (RAM)

Steps

a) Open file MMPL_UserIf.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.

2.7.4 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: **Move CRC tables into Code Memory (ROM)**

Steps

a) Open file MMPL_UserIf.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.

3 Making calls into MMPL-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
DoModbusTransaction	Periodically for every channel, when there is need to send request/response to/from the Modbus	Yes	–> This is the main entry point into the library, also called as the trigger function. –> DoModbusTransaction is a blocking call and does not return until a response

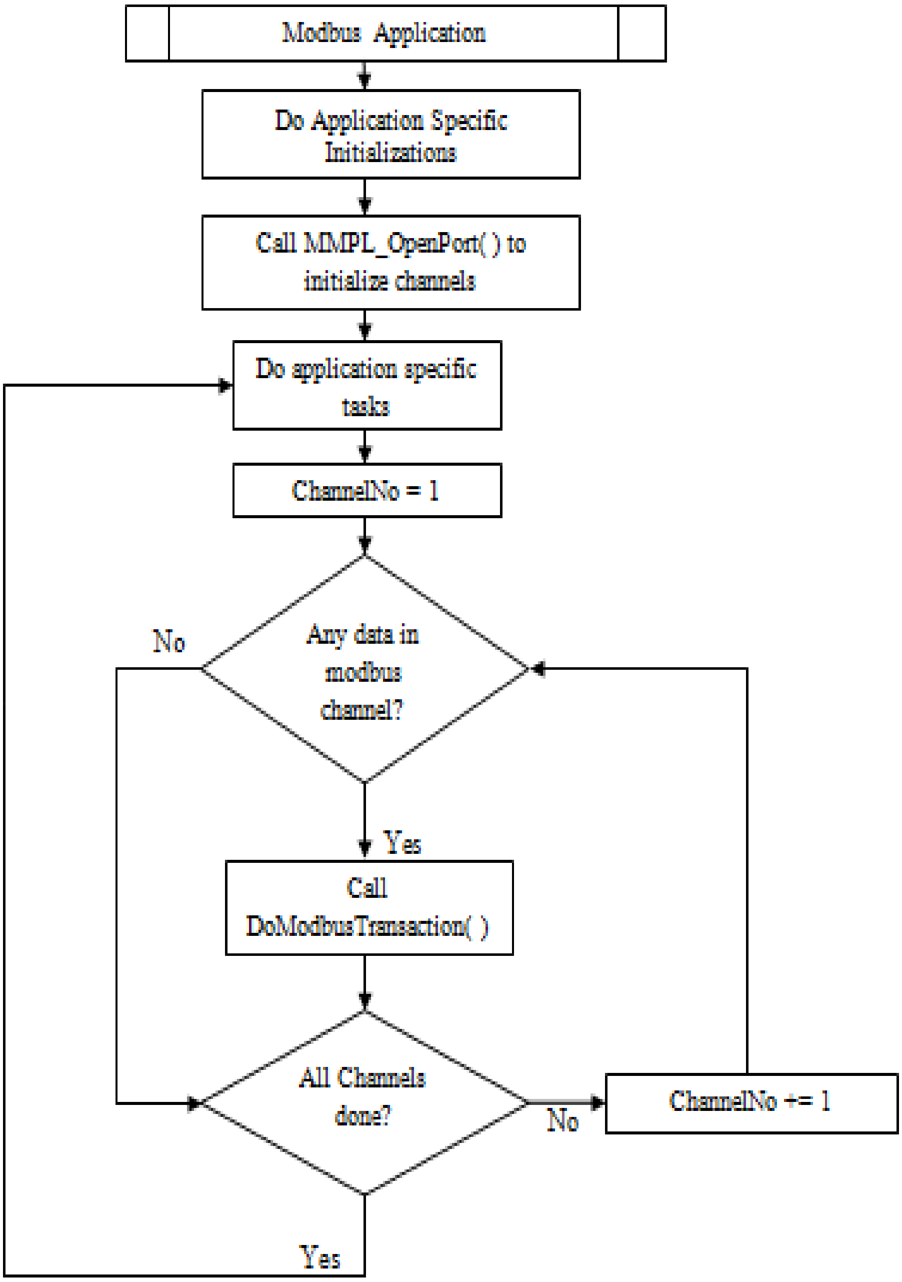
	Slave .		is received or a timeout occurs.. –> If DoModbusTransaction is used, then MMPL_ReadPort function in MMPL_UserIf.c must implement a timeout logic

MMPL_SendRequest/ MMPL_RecvAndProcessResp (These functions do what *DoModbusTransaction* single handedly does)	Periodically for every channel, when there is need to send request/response to/from the Modbus Slave .	Yes	–> *MMPL_SendRequest* function constructs and sends Modbus request
			–> *MMPL_RecvAndProcessResp* function receives and processes slave response
			–> Unlike *DoModbusTransastion*, the call does not get blocked waiting for slave to respond
			–> MMPL_SendRequest( );
			DoAnyOtherAppTass()/* Call any other application tasks*/
			if(ResponseReceived){
			MMPL_RecvAndProcessResp( );
			}

MMPL_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.
MMPL_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. –> As for *MMPL_OpenPort*, user may choose to implement channel de-initialisation code outside the library in which case this function need not be implemented or called.

3.1 Flowchart for MMPL-C API invocation

Diagram below shows a flowchart of invocation of the *MMPL_OpenPort* function and *DoModbusTransaction* function.


Modbus Application
Do Application Specific Initializations
Call MMPL_OpenPort( ) to initialize channels
Do application specific tasks
ChannelNo = 1
Any data in modbus channel?
No
Yes
Call DoModbusTransaction( )
All Channels done?
No
ChannelNo += 1
Yes

4 MMPL-C Reference

4.1 MMPL-C Data Types

These data types are defined in *MMPL_Defs.h*

MMPL-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;`

4.2 MMPL-C Function Reference

4.2.1 MMPL_OpenPort

Function name	MMPL_ OpenPort	
Description	This function should open communication port and initialize 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 initialized successfully or not.	
Possible return values	CSPL_TRUE - The specified port was opened and initialized successfully. CSPL_FALSE - The specified port could not be opened or initialized.	
Arguments	CSPL_U8 networkNo	A number identifying the "port" or channel used for Modbus communication that is to be initialized.
Called by	User application	
User Implements?	Yes	

4.2.2 MMPL_ClosePort

Function name	MMPL_ 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. 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	

4.2.3 MMPL_ReadPort

Function name	**MMPL_ 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*. 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	

4.2.4 MMPL_WritePort

Function name	MMPL_ 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*. 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	

4.2.5 MMPL_DebugPrint

Function name	MMPL_ 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_CHAR* debugMessage	A null-terminated 'C' string containing the debug message.
Called by	Library	
User Implements?	Yes	

4.2.6 MMPL_SendRequest

Function name	MMPL_SendRequest	
Description	MMPL_SendRequest function constructs and sends Modbus request	
Returns	**CSPL_U8**. A status code as shown in section below titled "**Status codes returned by function MMPL_SendRequest**"	
Arguments	MMPL_MB_REQ_ADU *pMbReqAdu	Used to hold the Modbus request ADU to be sent to the slave.
	MMPL_MB_RSP_ADU *pMbRspAdu	Used to hold the Modbus response ADU received from the slave.
	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.
	CSPL_U8 slaveNo	A single byte value containing the slave ID of the device from which data is being requested.
	CSPL_U8 functionfode	A single byte value of the Modbus function code that defines the Modbus service request.
	CSPL_U16 startAddress	A two-byte value that is the first address in the range of data being requested for.
	CSPL_U16 numItems	A two-byte value that is the number of data items starting from startAddress that are being requested for.
	CSPL_U8 *dataBuffer	-> (OUT): Pointer to an array of bytes into which the requested data must be copied into in the correct format for 'Read' FCs. -> (IN): Pointer to an array of bytes containing the data that has to be 'written' to slave.
Called by	User application	
User Implements?	No	

4.2.7 MMPL_RecvAndProcessResp

Function name	MMPL_RecvAndProcessResp	
Description	MMPL_RecvAndProcessResp function receives and processes slave response	
Returns	**CSPL_U8**. A status code as shown in section below titled "**Status codes returned by function MMPL_RecvAndProcessResp**"	
Arguments	MMPL_MB_REQ_ADU	Used to hold the Modbus request ADU to be sent to the slave.

	*pMbReqAdu	
	MMPL_MB_RSP_ADU *pMbRspAdu	Used to hold the Modbus response ADU received from the slave.
	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.
	CSPL_U8 slaveNo	A single byte value containing the slave ID of the device from which data is being requested.
	CSPL_U16 numItems	A two-byte value that is the number of data items starting from startAddress that are being requested for.
	CSPL_U8 *dataBuffer	-> (OUT): Pointer to an array of bytes into which the requested data must be copied into in the correct format for 'Read' FCs. -> (IN): Pointer to an array of bytes containing the data that has to be 'written' to slave.
Called by	User application	
User Implements?	No	

4.2.8 DoModbusTransaction

Function name	DoModbusTransaction	
Description	This method is the main function that drives Modbus communication on a network.	
Returns	**CSPL_U8**. A status code as shown in section below titled "Status codes returned by function DoModbusTransaction"	
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.
	CSPL_U8 slaveNo	A single byte value containing the slave ID of the device from which data is being requested.
	CSPL_U8 functionfode	A single byte value of the Modbus function code that defines the Modbus service request.
	CSPL_U16 startAddress	A two-byte value that is the first address in the range of data being requested for.
	CSPL_U16 numItems	A two-byte value that is the number of data items starting from startAddress that are being requested for.

	CSPL_U8 *dataBuffer	-> (OUT): Pointer to an array of bytes into which the requested data must be copied into in the correct format for 'Read' FCs. -> (IN): Pointer to an array of bytes containing the data that has to be 'written' to slave.
	CSPL_U8 numRetries	The number of times to retry communication with slave.
Called by	User application	
User Implements?	No	

4.2.9 Status codes returned by function DoModbusTransaction, MMPL_SendRequest and MMPL_RecvAndProcessResp

The following error codes may be returned by the main entry point function DoModbusTransaction, MMPL_SendRequest and MMPL_RecvAndProcessResp. They are defined in *MMPL_Defs.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 initialized with a call to MMPL_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 *MMPL_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

Error	Code	Remarks
		the library was trying to read from the channel.
INVALID_FC	0x0A	An invalid/unsupported function code was requested to be serviced.
TXID_MISMATCH	0x0B	The Transaction ID of the Modbus request does not match the response's Transaction ID.
INVALID_PROTCODE	0x0C	Invalid Protocol code in the response.
EXCEPTION_RESPONSE	0x0D	Exception response from slave.
FC_MISMATCH	0x0E	The function code of the Modbus request does not match the response's function code.
INVALID_BYTECNT	0x0F	Invalid Byte count in the response.
INVALID_DATA_VALUE	0x10	Invalid Data Value.
INVALID_PKTLEN	0x12	Invalid Packet Length in the response.
INVALID_SLAVE_ADDR	0x13	Invalid Slave ID.
INVALID_NUM_ITEMS	0x14	Invalid number of items.

4.3 Macro Reference

4.3.1 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.	

4.3.2 ENDIAN_STYLE

Macro name	ENDIAN_STYLE	
Description	Defines the Endian style of the processor running the library.	
Permitted	LITTLE_ENDIAN	Processor is of type Little Endian

Values	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.	

4.3.3 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	ι 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. ιι 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.	

4.3.4 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	ι 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. ιι 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.	

4.3.5 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	i 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. ii 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.

4.3.6 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	i 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. ii 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.	

4.3.7 RX_BUFFER_SIZE and TX_BUFFER_SIZE

Macro name	**RX_BUFFER_SIZE** **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	i 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. ii 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. iii The recommended way to set these macros is by using the MMPL configurator which automatically calculates the values for the buffers based on the values set for the block size limiting macros. iv 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)	

4.3.8 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	ι This macro is used by the library when creating debug messages. ιι 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.	

4.3.9 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	ι 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. ιι 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.	

4.3.10 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		
	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	ι This macro is used only in MODBUS RTU mode. ιι 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. ιιι 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. ιv The IN_ROM setting saves RAM (data memory) at the cost of using more code memory.	

4.3.11 CRC_TABLE_LOC_MODIFIER

Macro name	CRC_TABLE_LOC_MODIFIER	
Description	This macro is used to set the keyword that will cause variables in code memory (ROM) or RAM.	
Permitted Values		
	static xdata	Set value such that the table goes into RAM.
	code	Set value such that the table goes into ROM/Flash.
Remarks	ι Many compilers by default may store constant variables in code memory. If so, set this macro to a blank.	

4.3.12 xdata

Macro name	xdata
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	ι 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. ii 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. iii 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.