MODBUS Master Source Code Library

User Manual

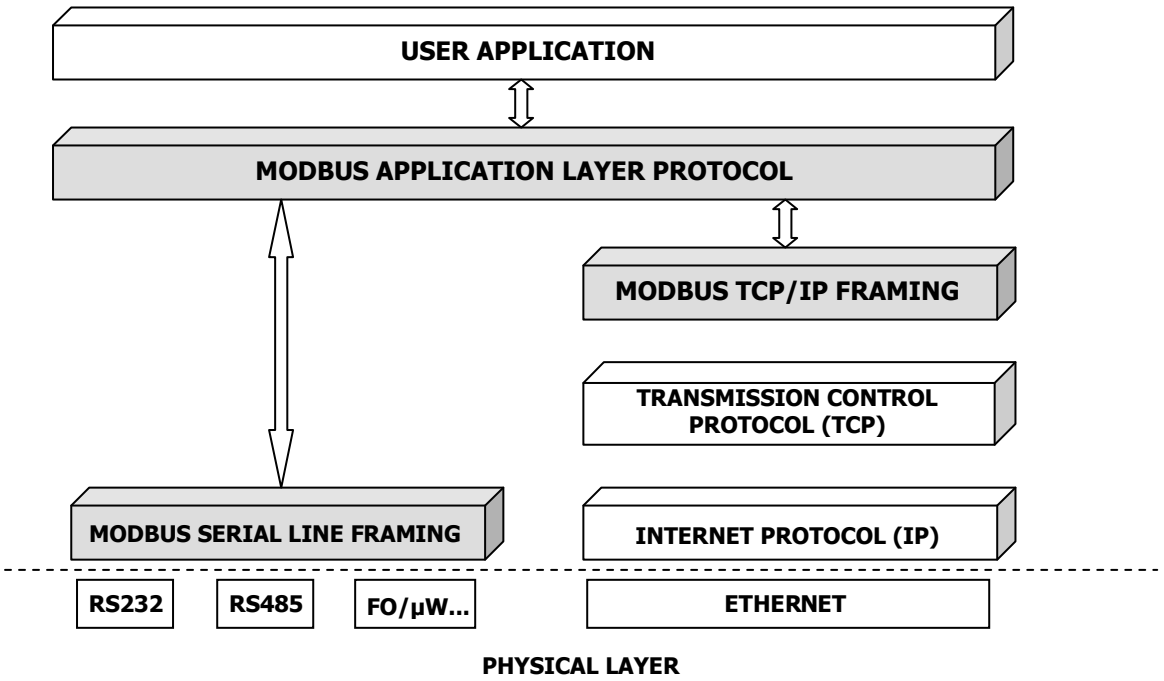
Version 2.1

Sunlux Technologies Ltd., No. 497, 6th 'A' Main, H I G Colony, R M V II Stage, Bangalore – 94, India.
Ph: ++91 80 3417072 Fax: ++91 80 3417072
Email: **info@sunlux-india.com** Web: **www.sunlux-india.com**

Table Of Contents

1.0 Introduction

MODBUS® Protocol is a messaging structure developed by Modicon in 1979, used to establish master-slave/client-server communication between intelligent devices. It is a de facto standard, truly open and the most widely used network protocol in the industrial manufacturing environment. MODBUS is an application layer messaging protocol, positioned at level 7 of the OSI model that provides client/server communication between devices connected on different types of buses or networks. MODBUS continues to enable millions of automation devices to communicate. Today, support for the simple and elegant structure of MODBUS continues to grow. The Internet community can access MODBUS at a reserved system port 502 on the TCP/IP stack. MODBUS is a request/reply protocol and offers services specified by function codes.



2.0 The MODBUS Master Stack Source Code Library

The MODBUS Master Source Code Library (SCL) is an attempt towards assisting Original Equipment Manufacturers and HMI Software vendors in quickly implementing MODBUS support into their devices/products. The Master SCL is an ANSI 'C' implementation of the MODBUS Master, which the OEM integrates and ports on to the native hardware. With the SCL the OEM can implement the MODBUS stack without having any knowledge of the MODBUS standard. The implementation follows strict ANSI 'C' standard to enable porting of the same on different kinds of platforms.

The figure above shows the Enhanced Protocol (Performance) Architecture (EPA) model of the MODBUS Protocol Stack. At the top of the layer is the User Application, which provides the MODBUS stack with the data to be sent in MODBUS frames. The MODBUS Application Layer Protocol below the top layer handles the task of assembling the required MODBUS frame based on requests. The MODBUS Serial Line framing layer adds the necessary error checking bytes to the MODBUS frame before transmitting it over the physical layer. The physical layer can be any asynchronous serial device like RS232, RS485, Fiber Optic, microwave etc. The MODBUS Serial Line implementation is the most commonly used standard today.

Another popular implementation of MODBUS is the MODBUS/TCP implementation. This implementation utilizes the popular TCP/IP stack over Ethernet as the delivery media. The MODBUS TCP/IP Framing layer handles the additional bytes to be prefixed with the MODBUS frame before handing it over to the TCP/IP layers, which eventually transmit the data over the Ethernet media.

The MODBUS Master SCL implements the blocks shown in gray background – the MODBUS Application Layer Protocol, the MODBUS Serial Line Framing Layer and the MODBUS TCP/IP Framing layer. The SCL provides interface Macros for the user to define using which the user can integrate the stack with his application on one side and the physical layer on the other. A following section describes in detail the procedure for porting the stack onto a different platform.

3.0 Important Concepts Used in SCL

There are some important terms like Network, Session Number, Slave number etc, which the user must know before using this SCL. The same is briefed out here for easy reference. For example consider a "Single Master & multiple slave network" as shown in the figure below. The Modbus slaves are connected to the Master PC through the serial communication ports COM1, COM2 and COM3. Then we say that the Modbus Master is connected to three serial networks. It is to be noted that 'n' number of networks are numbered as 0 to 'n-1'. Each of the networks has multiple number of slaves. In each network the slaves are numbered as 1, 2, 3…n. Hence two slaves on different networks may have the same slave number. The unique identifier that differentiates all the slaves (connected to the Modbus Master) from each other is the Session number. No two slaves connected to the Master can have the same Session number. Whenever the Modbus master communicates with one slave, a session is formed. Hence the slaves connected to the Master altogether are numbered as 1, 2, 3, ..n.

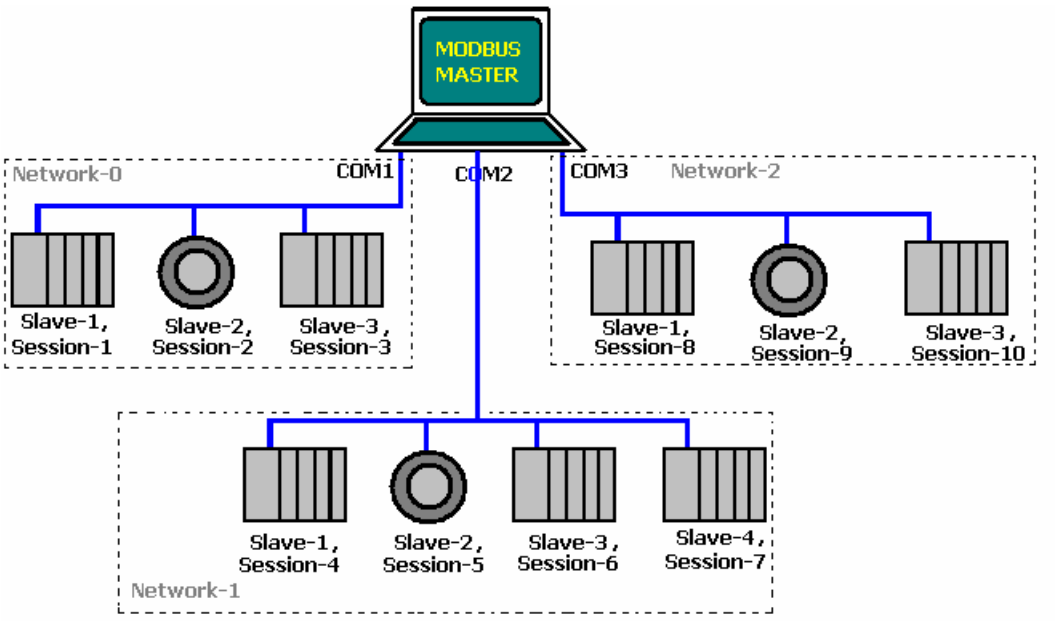


Figure illustrating Modbus Network

4.0 Pre-requisites

There are some pre-requisites before the SCL can be used in terms of information/knowledge and/or tools/techniques. The same is explained below:

4.0.1 Knowledge of ANSI 'C' programming

The SCL has been implemented fully with ANSI 'C'. Porting of the SCL to a specific platform requires the user to have ANSI 'C' programming knowledge since the porting activity involves implementation of some functions and definition of some Macros.

4.0.2 MODBUS Communication terminologies and techniques

A basic knowledge of what MODBUS is used for and how the communication takes place is useful. A brief discussion of the same is included at the end of this section.

4.0.3 'C' Compiler for the native platform

The platform on which the SCL is intended to be ported on must have a 'C' compiler since the entire SCL must be recompiled after the Macro definition and implementation.

4.1 MODBUS Basics

MODBUS is an application layer Master-Slave protocol used for transfer of data between two devices. The Master device always initiates a read/write request to which the Slave device responds. The Slave device never transmits anything on its own – it must be triggered with a request.

4.1.1 MODBUS Data types

There are four different kinds of data that MODBUS can transfer:

4.1.1.1 Coils – These are digital outputs. Coils can be read or written to. A possible value for Coils is either '0' or '1'.

4.1.1.2 Discrete Inputs – These are digital inputs. This kind of data can only be read and cannot be written to since they represent field inputs whose value is dependent of the field signals.

4.1.1.3 Holding Registers – Holding Register is a two-byte value (a WORD). Registers are used for storing analog values. A Holding Register is an analog output – it can be written to and read also.

4.1.1.4 Input Register – An input register is an analog input. Its value can be read but it cannot be written to for the same reason as for Digital Inputs.

4.1.2 MODBUS Device addressing

MODBUS is a multipoint protocol. This means that one Master can communicate with multiple slaves on the same communication line. Due to this a given slave must have a unique ID with which to address it – a MODBUS device address. A slave's device address MUST be unique on a given communication network – duplicate addresses lead to bus collision. MODBUS Device addresses must lie in the range 1 to 247. **The MODBUS Master Source Code Library does support broadcast addressing.** For broadcasting a message the Device address must be specified as 0. All the slaves on the specified network will receive the same request. It should be noted that broadcasting is not possible for all the requests. The user should know what all commands support the broadcasting, which is given in the section **10.0**. **The MODBUS Master Source Code Library can also handle exception responses got from the slave and will produce appropriate error messages, which is explained in section 8.2.**

4.1.3 MODBUS Data Point addressing

MODBUS uses unique addresses to point to data points. Each of the four data types has independent addresses starting from 0001 to FFFF. This means that there can be a Coil located at 0001 and a Digital Input also at address 0001. However a slave device need not necessarily have data points at all the addresses. The data points need not be at consecutive locations.

5.0 Components of the MODBUS Master Source Code Library

The MODBUS Master SCL is implemented using the following files:

5.0.1 **MBDefs.h** -- This file contains the type definitions of some basic data types and definitions of some symbolic literals.

5.0.2 **MBMaster.h** -- This file contains the macros, structures and function declarations used in the other files of SCL. The end user should not make modifications to this file.

5.0.3 **MBDriver.c** -- This file contains the implementation of the MODBUS Master Stack. It contains the functions used to initialize various data structures, functions to construct modbus data structures and to send and receive modbus frame. This file also contains the error handling code and declarations for the global variables created by the stack. The end user should not make modifications to this file.

5.0.4 **MBFrames.c** -- This file contains the functions for constructing & parsing the modbus frames. The end user should not make modifications to this file.

5.0.5 **MBLinkUser.h** – This is the file that contains the open Macros to be implemented by the end user. A sample declaration of all Macros is already made to ease the process for the end user. The user may change the declarations as required.

5.0.6 **MBLinkUser.c** – This file is a sample, basic implementation of the Macro definitions corresponding to the Macros declared in **MBLinkUser.h**. The implementation is very basic and is given for the purpose of demonstration only and as such is insufficient for the working of the stack.

6.0 Porting the Source Code Library

Since the SCL has been written using ANSI 'C' it is portable between operating systems and also between hardware platforms. The implementation is independent of physical transmission layer giving the user full freedom to choose the media. Based on the physical layer chosen, the physical layer Macros must be implemented by the user. Similarly the user interface of the driver (i.e. the manner in which the SCL interacts with the user application and database) is also left open – the user can implement it in any manner desired by him. So the porting of the SCL involves defining the Macros (i.e. mapping the Macros to actual function names) and implementing the functions.

The porting of the SCL to a native platform is done in three steps as below:

6.0.1 Define and implement the User Application Interface Macros and Functions
- 6.0.1.1 MBLINKUSER_GETTOTALNOOFNETWORKS
- 6.0.1.2 MBLINKUSER_GETNETWORKNO
- 6.0.1.3 MBLINKUSER_GETSLAVENO
- 6.0.1.4 MBLINKUSER_LINKCHANGESTATUS
- 6.0.1.5 MBTARG_GETSYSTIMEMILLISECS

6.0.2 Define and implement the Physical Layer Interface Macros and Functions
- 6.0.2.1 MBLINKUSER_INITCOMMNPATH
- 6.0.2.2 MBLINKUSER_CLOSECOMMNPATH
- 6.0.2.3 MBLINKUSER_READFROMCOMMNPATH
- 6.0.2.4 MBLINKUSER_WRITETOCOMMNPATH
- 6.0.2.5 MBLINKUSER_FLUSHCOMMNPATH

6.0.3 Define the Stack Control Macros
- 6.0.3.1 DEBUGENABLED
- 6.0.3.2 MODBUS_TCP
- 6.0.3.3 LITTLE_ENDIAN
- 6.0.3.4 BIG_ENDIAN

6.1 The User Application Interface Macros and Functions

The SCL implementation of the stack requires some information, which is user specific for it's functioning (e.g. number of slaves to communicate with). To enable the SCL obtain this information several User Application Interface Macros have been defined, which the end user is expected to implement. The stack calls these Macros to obtain some user specific data from the user application and also allow interaction between the stack and the user application. The following Macros are provided for the user to interface his application and database with the stack:

6.1.1 MBLINKUSER_GETTOTALNOOFNETWORKS ()

This Macro is called by the stack to determine the total number of networks present. As this SCL supports only single network this macro has been hardcoded to return 1. The end user need not make modifications to this macro. This macro is reserved for future use only.

Expected return value: BYTE (unsigned char), the total number of networks
Parameters: None

A sample implementation is as below:

```
// MBLinkUser.h file
#define MBLINKUSER_GETTOTALNOOFNETWORKS()   GetTotalNumOfNetworks()

// MBLinkUser.c file – returns the total num of networks
unsigned char GetTotalNumOfNetworks() {  return 1; }
```

6.1.2 MBLINKUSER_GETNETWORKNO(RequestSessionNum)

This Macro is called by the stack to determine the network number for a particular session number. Session number is the unique identification number for a slave. Two slaves on different networks can have the same address but they will have different session numbers. As this SCL supports only single network both session number and slave number will be same. Hence this macro has been hardcoded to return 0. This macro is reserved for future use only. More information about Network number and Session number is given in section-3.0. It should be noted that if NumOfNetworks is 'x', then 0 to 'x-1' networks are available. The end user need not make modifications to this macro.

Expected return value: unsigned char, the network number for given slave number
Parameters:

1. *unsigned char RequestSessionNum*

A sample implementation is as below:

```
// MBLinkUser.h file
#define MBLINKUSER_GETNETWORKNO()  GetNetworkNum(RequestSessionNum)

// MBLinkUser.c file – returns zero since first network is referenced as Network number zero.
unsigned char GetNetworkNum (RequestSessionNum) { return 0; }
```

6.1.3 MBLINKUSER_GETSLAVENO(RequestSessionNum)

This Macro is called by the stack to determine the slave number for a particular session number. Session number is the unique identification number for a slave. Two slaves on different networks can have the same MODBUS address but they will have different session numbers. As this SCL supports only single network both session number and slave number are presently same. Hence this macro has been defined to return the session number itself. This macro is reserved for future use only. The end user need not make modifications to this macro.

Expected return value: unsigned char, the slave number for given session number
Parameters:

1. *unsigned char RequestSessionNum*

A sample implementation is as below:

```
// MBLinkUser.h file – actual function name is GetSlvNo
#define MBLINKUSER_GETSLAVENO(RequestSessionNum)  GetSlvNo(RequestSessionNum)

// MBLinkUser.c file – returns the session number
unsigned char GetSlvNo(unsigned char RequestSessionNum) {  return(RequestSessionNum); }
```

6.1.4 MBLINKUSER_LINKCHANGESTATUS(NetworkNo, Status)

This Macro is called by the stack to notify the end user when the status of the network changes. It is called when a network goes from online(GOOD) to offline(BAD) state or from offline to online state. Changed State of the network is passed in the parameter 'Status'. When the stack tries to read or write from/to the communication path and the operation fails due to the communication path error the stack changes the network status from online(GOOD) to offline(BAD) state. Communication path error is one which is caused by the failure of com port. Once the network status is changed to offline(BAD) state no further network read/write operations are possible. To make the network status online(GOOD) again the function 'MBDriver_Init' (described in section 9.1) must be called. In the SCL this macro has been hard coded to 1. The end user may implement a function to notify the MMI and may take the appropriate action.

Expected return value: None
Parameters:

1. *unsigned char NetworkNo, The network number*
2. *unsigned char Status, 1 for good, 0 for bad*

A sample implementation is as below:

```
// MBLinkUser.h file
#define MBLINKUSER_LINKCHANGESTATUS(NetworkNo, Status)
                                        NetworkStatusChange(NetworkNo, Status)
```

```
// MBLinkUser.c file – Calls a function to notify the MMI about the change of network status.
void NetworkStatusChange(NetworkNo, Status) {  FunctionToNotifyMMI(); }
```

6.1.5 MBTARG_GETSYSTIMEMILLISECS()

This Macro is called by the stack to determine the system time in milliseconds. The Macro should return the system time in milliseconds. Native function call on the target OS for getting system time in milliseconds should be given. The stack uses this macro for time-out processing. The stack gets the system time just before transmitting the request to the slave. Then it keeps on calling this macro till the response from the slave is obtained. Every time it compares the transmitted time and the presently obtained time. If the slave doesn't responds within the specified Timeout period the stack generates the 'Timeout' error.

Expected return value: Unsigned long, System time in Milliseconds.

Parameters: None

A sample implementation is as below:

```
// MBLinkUser.h file – actual function name is GetTimeInMilliseconds
#define MBTARG_GETSYSTIMEMILLISECS ()  GetTimeInMilliseconds()

// MBLinkUser.c file – returns the system time in milliseconds
unsigned long GetTimeInMilliseconds()  {
        time_t ltime;
        time(&ltime);
        return ((unsigned long)ltime * 1000);  }
```

6.2 Physical Layer Interface Macros and Functions

The implementation of the SCL stops at the point of framing and parsing the MODBUS data – the functionality of transmitting and receiving the frames to/from the physical layer is kept open to enable portability of the SCL to different platforms and also to different physical layers. These Macros allow the stack to use the Physical Layer chosen by the user to transmit and receive MODBUS frames. The following Macros are available:

6.2.1 MBLINKUSER_INITCOMMNPATH(PortNo, BaudRate, Parity, DataBits, StopBits)

This Macro is called by the stack during the application startup to initialize its communication path. This Macro is expected to return a path ID or handle to the communication path on which further MODBUS communication is to happen. The path ID/handle will have different forms in different operating systems/platforms. For e.g. on a UNIX platform a serial communication path identifier is a simple two byte value representing the path. On Windows it is a double word Handle to the serial communication port. The form is different for a TCI/IP Socket connection. So based on the desired physical interface and the port this function must return a unique communication path identifier, which will be used for all further communications. The option of setting up the communication path by initializing it with various parameters should be done in this Macro itself. A sample implementation for a Windows based application can be as below:

Expected Return Value:

Unsigned short NetworkId– a unique identifier to the comm. Path
0 – If Initialization fails

Parameters: These parameters are the same, which are supplied by the user application through the function 'MBDriver_Init ()' .

1. unsigned char PortNo
2. unsigned long BaudRate
3. unsigned char Parity
4. unsigned char DataBits
5. unsigned char StopBits

A sample implementation is as below:

```
// MBLinkUser.h file – actual function name is Open()
#define  MBLINKUSER_INITCOMMNPATH (PortNo, BaudRate, Parity, DataBits, StopBits)
                                        Open (PortNo, BaudRate, Parity, DataBits, StopBits)

// MBLinkUser.c file – a Windows implementation for a serial communication port
unsigned short Open(unsigned char PortNo, unsigned long BaudRate, unsigned char Parity,
                                        unsigned char DataBits, unsigned char StopBits)
{
        DCB dcb; COMMTIMEOUTS TimeOut; BOOL fSuccess;
        BOOL bReturn = TRUE; HANDLE hCom;
        char strPort[10];
        sprintf(strPort, "COM%d", PortNo +1);
        hCom = CreateFile(strPort, GENERIC_READ | GENERIC_WRITE,
                0,    /* comm devices must be opened w/exclusive-access */
                NULL, /* no security attrs */
                OPEN_EXISTING, /* comm devices must use OPEN_EXISTING */
                0,    /* not overlapped I/O */
                NULL  /* hTemplate must be NULL for comm devices */ );

        if (hCom == INVALID_HANDLE_VALUE)  { bReturn = FALSE; }

        fSuccess = GetCommState(hCom, &dcb);
        if (!fSuccess)  { bReturn = FALSE; }

        dcb.DCBlength = sizeof(DCB); dcb.BaudRate = BaudRate; dcb.ByteSize = DataBits;
        dcb.Parity = Parity; dcb.StopBits = StopBits;

        fSuccess = SetCommState(hCom, &dcb);
         if (!fSuccess) { bReturn = FALSE; }

        fSuccess = GetCommTimeouts(hCom,&TimeOut);
        if (!fSuccess) { bReturn = FALSE; }

        TimeOut.ReadTotalTimeoutConstant = 0;
        TimeOut.ReadTotalTimeoutMultiplier = 0;
        TimeOut.ReadIntervalTimeout = MAXDWORD;

        fSuccess = SetCommTimeouts(hCom,&TimeOut);
         if (!fSuccess)  { bReturn = FALSE; }

        return(bReturn ? (WORD)hCom : FALSE);
}
```

6.2.2 MBLINKUSER_CLOSECOMMNPATH(NetworkId)

The stack calls this macro just before it exits, to close and free the communication path/port used by the stack for MODBUS communication. This allows other applications to use the communication path after the MODBUS stack/user application exits. The implementation must de-initialise the communication path if required and then close it.

Expected Return Value: TRUE if successful, FALSE otherwise
Here TRUE=1 & FALSE=0

Parameters:

1. *unsigned short NetworkId : A unique ID returned by the function* **'INITCOMMNPATH'** *representing the communication path, which is to be closed.*

A typical Windows implementation is as below:

```
// MBLinkUser.h file – actual function name is Close()
#define MBLINKUSER_CLOSECOMMNPATH(NetworkId)        Close(NetworkId)

// MBLinkUser.c file
unsigned char Close(NetworkId){ return CloseHandle( (HANDLE) NetworkId); }
```

6.2.3 MBLINKUSER_READFROMCOMMNPATH(NetworkId, ReadBuf, NoOfBytesToRead, NoOfBytesRead)

The stack calls this macro to read the characters/bytes from the communication path. This macro must be implemented for the stack to function properly. The implementation of this macro must read the requested number of bytes from the specified communication path and copy the same into the pre-allocated buffer which is passed as a parameter. The function should also copy the number of bytes of data read into a pointer variable which is passed as a parameter. If an error is encountered in the reading process (for e.g. due to network fault), the function should return FALSE(zero value) indicating failure. Otherwise even if the number of bytes read is zero (for e.g. when there are no bytes available at the network/port) the function should return TRUE (i.e. value 1). It should be noted that if this macro returns FALSE the Network status will be marked as Offline(BAD) as explained in section 6.1.4. The "timeout" processing logic is applied within the stack for this function. Hence there is no need to implement timeout processing in the macro function definition.

Expected Return Value: TRUE if No error occurs during reading
FALSE otherwise
Here TRUE=1 & FALSE=0

Parameters:

1. *usigned short NetworkId – A unique ID returned by the function* **INITCOMMNPATH'** *representing the communication path.*
2. *unsigned char * ReadBuf – A pre-allocated buffer to store received bytes.*
3. *unsigned long NoOfBytesToRead – Num of bytes to read from path.*
4. *unsigned long * NoOfBytesRead – Pointer variable in which the number of bytes read to be put.*

A typical Windows implementation is as below:

```
// MBLinkUser.h file
#define  MBLINKUSER_READFROMCOMMNPATH(NetworkId, ReadBuf, NoOfBytesToRead,
  NoOfBytesRead)        Read(NetworkId, ReadBuf, NoOfBytesToRead, NoOfBytesRead )

// MBLinkUser.c file – a Windows implementation for a serial communication port with
//no timeout processing
unsigned char Read(unsigned short NetworkId, unsigned char *ReadBuf,
                    unsigned long NoOfBytesToRead, unsigned long *NoOfBytesRead )
{
MBDEFS_BOOL bSuccess = ReadFile((HANDLE)NetworkId, ReadBuf, NoOfBytesToRead,
                                                              NoOfBytesRead, NULL);
            return bSuccess;
}// end of function
```

6.2.4 MBLINKUSER_WRITETOCOMMNPATH(NetworkId, RequestFrameBuf, NoOfBytesToWrite, NoOfBytesWritten)

The stack calls this macro to write the MODBUS reply to the communication path. This macro must be implemented for the stack to function properly. The implementation of this macro must write the requested number of bytes from the buffer (passed as a parameter) to the specified communication path and should also put the number of bytes of data actually written into a pointer variable (passed as a parameter). If an error is encountered in the writing process(for e.g. due to network fault), the function should return FALSE(zero value) indicating failure. Otherwise the function should return TRUE (i.e. value 1). It should be noted that if this macro returns FALSE the Network status will be marked as Offline(BAD) as explained in section 6.1.4.

Expected Return Value: TRUE if No error occurs during reading
FALSE otherwise
Here TRUE=1 & FALSE=0

Parameters:

1. *unsigned short NetworkId – A unique ID returned by the function* **INITCOMMNPATH'** *representing the communication path.*

2. *unsigned char * RequestFrameBuf – A pre-allocated buffer containing the data to be written to the port.*
3. *unsigned long NoOfBytesToWrite – Number of bytes to write to path.*
4. *unsigned long * NoOfBytesWritten – Pointer variable in which the number of bytes written to be put.*

A typical Windows implementation is as below:

```
// MBLinkUser.h file
#define MBLINKUSER_WRITETOCOMMNPATH(NetworkId, RequestFrameBuf,
NoOfBytesToWrite, NoOfBytesWritten)      Write(NetworkId, RequestFrameBuf,
NoOfBytesToWrite, NoOfBytesWritten)

// MBLinkUser.c file – a Windows implementation for a serial communication port
unsigned char Write(unsigned short NetworkId, unsigned char *WriteBuf,
                        unsigned long NoOfBytesToWrite, unsigned long *NoOfBytesWritten )
{
        unsigned char bSuccess = WriteFile((HANDLE)NetworkId, WriteBuf,
                                                NoOfBytesToWrite, NoOfBytesWritten, NULL);
        return bSuccess;
}// end of function
```

6.2.5 MBLINKUSER_FLUSHCOMMNPATH(NetworkId)

The stack calls this macro to clear the communication buffer (i.e. delete residual bytes) whenever it encounters an error in reading the MODBUS frame so that it can start afresh with the next MODBUS frame. The implementation of this macro must clear all data of the communication path.

Expected Return Value: None

Parameters:

unsigned short NetworkId – A unique ID returned by the function **'INITCOMMNPATH'** representing the communication path whose buffer is to be cleared.

A typical Windows implementation is as below:

```
// MBLinkUser.h file
#define   MBLINKUSER_FLUSHCOMMNPATH(NetworkId)       Flush(NetworkId)

// MBLinkUser.c file – a Windows implementation for a serial communication port
 void Flush(unsigned short NetworkId)
        { PurgeComm((HANDLE)NetworkId,PURGE_TXCLEAR|PURGE_RXCLEAR); }
```

6.3 Stack Control Macros

The Stack Control Macros control the behavior of the stack by changing the control flow by means of pre-processor directives. By setting appropriate values for these macros the user can control the compilation process. The following macros are provided:

6.3.1 DEBUGENABLED

For ease of debugging the SCL uses several "printf" statements internally. However since these debugging statements consume considerable amount of code memory and also take up a lot of execution time, the DEBUGENABLED macro has been provided using which the user can enable or disable printing of debugging statements. To enable debugging define the above symbol – else comment it out.

Sample Example:

```
// MBLinkUser.h file – Enables the "printf" statements in the stack.
#define DEBUGENABLED
```

6.3.2 MODBUS_TCP

This symbol must be defined if the MODBUS stack is being used for communication over TCP/IP. The framing methodology followed for a serial line MODBUS communication and TCP/IP MODBUS communication is different. The MODBUS/TCP Specifications are different from the Serial standard mainly in two ways (1) Serial uses a two byte CRC at the end of the frame whereas MODBUS/TCP does not (since error checking and correction are handled in other layers of TCP/IP itself) (2) MODBUS/TCP uses an additional 6 byte header before the slave address field of a frame.
e.g.

```
// MBLinkUser.h file – Chooses the communication to take place over TCP/IP
#define MODBUS_TCP  1
```

6.3.3 LITTLE_ENDIAN

If the hardware architecture chosen follows the LITTLE_ENDIAN style of storing WORDs, then this symbol must be defined. Depending on the operating system and the hardware platform, there are two ways of storing a WORD variable in memory. In the first case called the BIG ENDIAN style the high byte of the WORD is stored first and then the low byte of the word. This style can generally be found in MOTOROLA based CPU architectures. In the other style called the LITTLE ENDIAN, the low byte of the WORD is stored first and then the high byte of the word. This style is more prominent in CPU's following the INTEL architecture. MODBUS follows the BIG ENDIAN style of packing WORDs in its frames. So on platforms supporting LITTLE ENDIAN style the two bytes forming the WORD must be reversed if the data is to be copied properly. This is internally done in the stack if the LITTLE_ENDIAN symbol is defined.

Sample Example:

```
// MBLinkUser.h file    //Selects the system on which the stack is running as Little_Endian
#define LITTLE_ENDIAN 1
```

6.3.4 BIG_ENDIAN

If the hardware architecture chosen follows the BIG_ENDIAN style of storing WORDs, then this symbol must be defined.

Sample Example:

```
// MBLinkUser.h file  //Indicates that system on which the stack is running is not Big_Endian
#define BIG_ENDIAN  0
```

NOTE : It should be noted that BIG_ENDIAN and LITTLE_ENDIAN are mutually exclusive. The end user is supposed to define these macros properly. For example if the hardware architecture chosen follows the LITTLE_ENDIAN style then the macros should be defined as shown in the examples above. If accidentally both the macros are defined to 1 then results may be unpredictable.

7.0 List of Global Variables

The stack uses global variables to optimize the execution of the code. In order to avoid errors due to accidental redefinition of these global variables in the user application all variables are prefixed with 'MB'. Please note that the list below is given only to avoid the user using same name as a global variable in his variable declarations and as such the user should not modify the variable names. Doing so may affect the working of the SCL.

1) NETWORK_INFO* MB_NetworkInfo; /* Array of Structures to hold the network details of each network */
2) MBDEFS_UCHAR MBErrNo ; /* Global variable to hold the error number */
3) MBDEFS_UINT8 MBSendBuffer[256];/*Buffer which contains the frame to be transmitted to slave*/
4) MBDEFS_UINT8 *MBReceiveBuffer; /*Buffer which contains the frame received from the slave*/
5) FRAMEFUNCTIONPTR g_FrameConstructFunctionArray[] /*Array of function pointers to construct and parse the Modbus frames*/
6) EXCEPTIONFUNCTIONPTR g_FrameExceptionFunctionArray[] /* Array of function pointers to handle the exception responses from the slave */

8.0 MODBUS Error checking and other information

The SCL stores error information on the error it encounters in a global variable named 'MBErrNo' of type unsigned char. The user application can read this variable to check which error occurred during the last MODBUS request handling. The possible error codes are defined in the MBErr.h file and are reproduced below for easy reference.

8.1 General Errors:

In case of General errors appropriate actions should be taken as specified below, specially in case of error number 17.

Error No.	MACRO	Description
0		No error. Successful execution of the command.
10	NOMEMORY	No Memory. Memory allocation operation has failed.
11	UNKNOWN_REQUEST	Unknown request. The value supplied by the user in the "Type" variable is other than 1,2,3,4. (See section 9.2 and 9.3.
12	INVALID_ADDRESS	Invalid address.
13	EXCEED_NO_OF_ITEMS	Number of items requested to read or write is exceeding the maximum limit.
14	TIMEOUT	Time out. Master has sent the request and no reply has come back.
15	CHECKSUM_ERROR	Checksum Error. Master has received the reply from the slave. But the integrity of the received data is incorrect.
16	SLAVEADD_MISMATCH	Slave address mismatch. Master has received the reply. But the slave number field in the received frame doesn't matches with that sent by the Master.
17	BAD_NETWORK	One of the Network operations (Read/Write) failed due to communication path error. The network status is marked as bad in the stack. The communication path should be checked out. In this case the Port has to be closed using the MBDriver_DeInit function and again it has to be initialized using MBDriver_Init function.
18	INVALID_SLAVE_NO	Invalid slave number entered by the user. Valid slave numbers are: For read functions 1 to 247. For write functions 0 to 247,where 0 is for broadcasting.

8.2 Exception Responses:

Exception Responses are the responses sent by the slave for a master request when the particular slave is unable to process the requested task. These errors are got only when the slaves being queried support this functionality. For example some slaves, which don't support this functionality may not send any response during such conditions. Then the master will process time out and will send time out error. The present stack is implemented to receive such exception responses and store the exception response code in the global variable 'MBErrNo'. The end user should note that no action is taken inside the stack after receiving the exception responses except for Exception Code -5. The entry point functions of the stack simply store the exception response number in the global variable and return false. The end user should take appropriate actions depending on the exception response. The possible exception responses are defined in the 'Modbus.h' file and are reproduced below for easy reference.

Error No.	**MACRO**	**Description**
1	ILLEGALFUNCTION	Illegal Function. The slave doesn't support the requested function
2	ILLEGALDATAADDRESS	Illegal data address. The data address received in the query is not an allowable address for the slave.
3	ILLEGALDATAVALUE	Illegal data value. A value contained in the query data field is not an allowable value for the slave.
4	SLAVEDEVICEFAILURE	Slave device failure. An unrecoverable error occurred while the slave was attempting to perform the requested action.
5	ACKNOWLEDGE	Acknowledge. The slave has accepted the request and is processing it. But a long duration of time will be required to do so. This response is returned to prevent a time out error from occurring in the master. The stack is implemented to wait again for the Slave response after receiving this exception code.
6	SLAVEDEVICEBUSY	Slave device busy. The slave is engaged in processing a long–duration program command. The master should retransmit the message later when the slave is free.
8	NEGATIVEACKNOWLEDGE	Negative Acknowledge. The slave cannot perform the program function received in the query.
9	MEMORYPARITYERROR	Memory Parity Error. The slave attempted to read extended memory, but detected a parity error in the memory. The master can retry the request, but service may be required on the slave device.

9.0 PROTOCOL Entry Functions

The end user needs to call only the protocol entry functions in his application program to avail the functionality of the protocol. There are Four entry functions which the user can make use. The Modbus driver must be initialized by calling 'MBDriver_Init' function before calling any other function. Once initialized the user can make use of 'MBRead' and 'MBWrite' functions for Modbus transactions. These functions construct the Modbus frame, send it to the specified slave and receive the response. Using these two functions the user can send the commands FC01 to FC06, FC15 & FC16 as specified in the section 8.0. At the end of the Modbus transactions 'MBDriver_DeInit' function should be called. An example of such a call is as given below:

```
void main(void)
{    unsigned char ErrNo, Retvalbool, PortNo=0, Parity=0;
     unsigned char DataBits=8, StopBits=0, SlaveNo=1; Retries=1,Type=1;
     unsigned short VarAddress=1, NItems=2;
     short *ReadData; unsigned long BaudRate=9600, TimeOut=1000;

     ReadData = (short *)malloc(2 * sizeof(short));
     Retvalbool=MBDriver_Init (PortNo, BaudRate, Parity, DataBits, StopBits);
     if(Retvalbool)
     { printf("\n init success\n");
       Type = 3 ; /* holding registers */
       Retvalbool=MBRead (SlaveNo, Type, VarAddress, NItems, ReadData, TimeOut, Retries);
                    if(Retvalbool)
                    {          printf("\n send &receive  success\n");
                               for(i=0;i<2;i++)  printf("data[%d]=%d\n",i,data[i]);
                    }
                    else       printf("\n send &receive  failure\n");
      }   free (ReadData);
          printf("%d\n",MBErrNo);     /* MBErrNo is Global variable and contains the error code */
     MBDriver_DeInit();     /*De initialize the stack before end of main function*/
} /*end of main*/
```

The prototypes of the entry functions and the files in which these functions are defined are described below.

9.1 MBDriver_Init (PortNo, BaudRate, Parity, DataBits, StopBits)

This function initializes the specified communication path with the specified parameters and it also initializes some data structures required by the stack. This function should be called before calling any other functions. The parameters are as explained below:

- **File name –** MBDriver.c
- **Parameters**

Sl. No.	Parameter Name	Description	Data Type
1	PortNo	Network or Port over which to open the Modbus communication.	unsigned char
2	BaudRate	Baud Rate for data transfer	unsigned long
3	Parity	Parity checking type for data transfer	unsigned char
4	DataBits	Byte size for data transfer	unsigned char
5	StopBits	Stop Bits for data transfer	unsigned char

- **Return Value** – unsigned char (BYTE)
  Return value of '1' indicates successful completion of the operation
  Return value of '0' indicates an error.

9.2 MBRead (SlaveNo, Type, VarAddress, NItems, data, Timeout, Retries)

This function is used to send the 'read commands' (FC01 to FC04) and receive the response from the Modbus slave. Note that since this version supports only single network, the session number and the slave number will be same.

- **File name –** MBDriver.c
- **Parameters**

Sl No	Parameter Name	Description	Data Type	Values
1	SlaveNo	Slave ID from which data is to be read	unsigned char	1 – 247
2	Type	Type of data to be requested from the slave	unsigned char	1 – Coil 2–DiscreteInput 3 – Holding Reg 4 – Input Reg
3	VarAddress	Starting address of the data to be read	unsigned short	1 – 65535
4	Items	No of items of the specified type to be read	unsigned short	**Number of Coils / Discrete-Inputs / Registers.
5	Data	Memory allocated data buffer for receiving the requested data	short*	Valid data buffer with memory allocated for receiving 'Items'
6	TimeOut	Time out in milliseconds for the Read operation	unsigned long	As decided by the end user.
7	Retries	No of retries in case of transmission failure	unsigned char	As decided by the end user.

- Return Value – unsigned char (BYTE)
  Return value of '1' indicates successful completion of the operation.
  Return value of '0' indicates an error. In case of an error refer to the error code in the global variable 'MBErrNo'.

****For Serial communication:**
Number of coils/Discrete inputs: 1 to 2008
Number of Registers: 1 to 125

For TCP/IP communication:
Number of coils/Discrete inputs: 1 to 1976
Number of Registers: 1 to 123

9.3 MBWrite (SlaveNo, Type, VarAddress, NItems, data, TimeOut, Retries)

This function is used to send the 'write commands' (FC05, FC06, FC15 & FC16) and receive the response from the Modbus slave.

- **File name –** MBDriver.c
- **Parameters**

Sl	Parameter Name	Description	Data Type	Values
1	SlaveNo	Slave ID to which data is to be written	unsigned char	0 – 247
2	Type	Type of data to be written to the slave	unsigned char	1 – Coil 3 – Holding Reg
3	VarAddress	Starting address of the data to be written	unsigned short	1 – 65535
4	Items	No of items of the specified type to be written	unsigned short	**Number of Coils / Discrete-Inputs / Registers.
5	Data	Data buffer containing the data to be written	short*	Valid data buffer containing 'Items' to be written.
3	TimeOut	Time out in milliseconds for the Write operation	unsigned long	As decided by the end user.
4	Retries	No of retries in case of transmission failure	unsigned char	As decided by the end user.

- Return Value – unsigned char (BYTE)
  Return value of '1' indicates successful completion of the operation.
  Return value of '0' indicates an error. In case of an error refer to the error code in the global variable 'MBErrNo'.

****For Serial communication:**
Number of coils/Discrete inputs: 1 to 1976
Number of Registers: 1 to 123
For TCP/IP communication:
Number of coils/Discrete inputs: 1 to 1944
Number of Registers: 1 to 121

9.4 MBDriver_DeInit ()

This function closes the communication path and deinitializes the data structures, which were used by the stack. This function has to be called before closing the user application.

- **File name –** MBDriver.c
- **Parameters** – None
- **Return Value** – unsigned char (BYTE)
  Return value of '1' indicates successful completion of the operation.
  Return value of '0' indicates an error.

10.0 Technical Specifications

Parameter	Value
Standard	MODBUS Application Protocol Specification V1.1, Nov 2002, Schneider Electric, www.modbus.org *Other references:* 1. MODBUS over Serial Line Specification & Implementation guide V1.0, Nov 2002, www.modbus.org 2. Modicon Modbus Protocol Reference Guide, PI–MBUS–300 Rev. J, June 1996, MODICON Inc.
Functions Supported	Read Coils (FC 01) Read Input Discretes (FC 02) Read Multiple Registers (FC 03) Read Input Registers (FC 04) *Write Single Coil (FC 05) *Write Single Register (FC 06) *Force Multiple Coils (FC 15) *Write Multiple Registers (FC 16)
Porting Methodology	User Definable 'C' Macros 1) For User Database Interface 2) For Physical Layer Interface
Development Language	ANSI 'C'
Supported Operating Systems	Portable to any 'C' supporting platform

*These Functions support Broadcasting.