MODBUS Slave Source Code Library

User Manual

Version 2.7

July 2006, Copyright Sunlux Technologies Ltd., All rights reserved.
Sunlux Technologies Ltd., No. 497, 6th ‘A’ Main, H I G Colony, R M V II Stage, Bangalore – 94, India.
Ph: ++91 80 23417073 Fax: ++91 80 23417073
Email: info@sunlux-india.com Web: www.sunlux-india.com

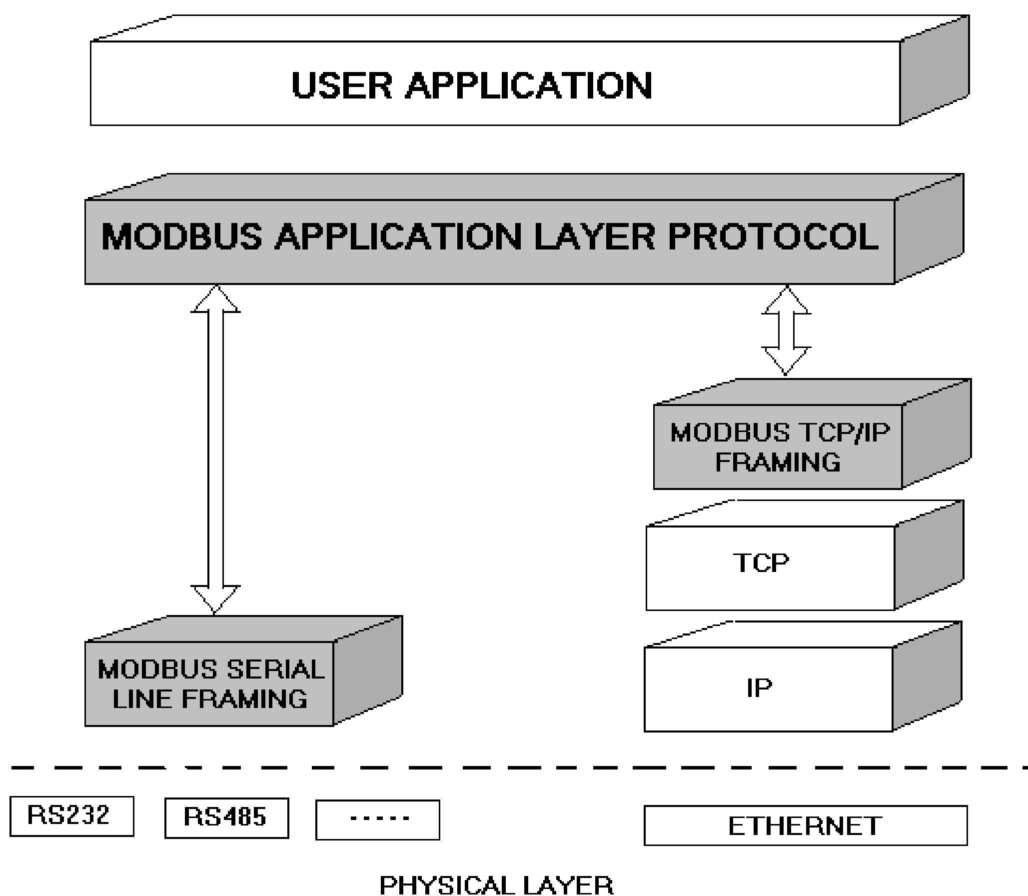
Table of Contents

Introduction..4
1.0)The MODBUS Slave Stack Source Code Library..5
2.0)Pre-requisites...6
2.1)MODBUS Basics...7
2.1.1)MODBUS Data types...7
2.1.2)MODBUS Device addressing...7
2.1.3)MODBUS Data Point addressing..7
3.0)Components of the MODBUS Slave Source Code Library...............................8
4.0)Porting the Source Code Library...9
4.1)The User Application Interface Macros and Functions.................................. 11
4.1.1)GETSLAVEADDR( ).. 11
4.1.2)CHECKADDRESSES(StartAddress, NoOfRegisters, DataType) 11
4.1.3)GETDATA(StartAddress, NoOfRegisters, Buffer, DataType)............................... 12
4.1.4)PUTDATA(StartAddress,NoOfRegisters,Buffer, DataType) 13
4.1.5)GET_EXCEPTION_COIL_DATA (CoilStsBuffer)... 14
4.1.6)GET_DEVICE_SPECIFIC_DATA (DevSpecsBuf, DataSize)............................ 14
4.1.7)GET_GENERAL_REF (FileNumber, StartAddress, RegCount, DataBuf)......... 15
4.1.8)PUT_GENERAL_REF (FileNumber, StartAddress, RegCount, DataBuf)..........15
4.1.9)READ_DEVICE_IDENTIFICATION (RDIReadDevId, RDIObjectId, RDIConformityLevel, RDIMoreFollows, RDINextObjId, RDINumOfObjs, RDIObjBuffer, RDIBufferSize).......16
4.1.10)RESTART_COMMUNICATIONS ( ).. 19
4.1.11)GET_DIAGNOSTIC_REG_VAL (DiagRegVal).. 20
4.1.12)SLAVE_ENTERS_LISTEN_ONLY_MODE ( )..20
4.1.13)CLEAR_COUNTERS_AND_DIAGNOSTIC_REGISTER ( )............................. 21
4.1.14)CLEAR_OVERRUN_COUNTER_AND_FLAG ( )... 21
4.1.15)READFIFO (FIFOPntrAdd, FIFOCount, FIFORegBuf)................................... 22
4.2)Physical Layer Interface Macros and Functions...23
4.2.1)GETCOMMNPATHNO().. 23
4.2.2)INITCOMMNPATH(CommnPathNo)... 23
4.2.3)READFROMCOMMNPATH(CommnPathNo, nBytes, Buffer)............................. 24
4.2.4)WRITETOCOMMNPATH(CommnPathNo, nBytes, Buffer)................................. 24
4.2.5)FLUSHBUFFER(CommnPathNo)... 25
4.2.6)CLOSECOMMNPATH(CommnPathNo).. 26
4.3)Stack Control Macros...27
4.3.1)LITTLE_ENDIAN.. 27
4.3.2)MODBUS_TCP.. 27
4.3.3)MODBUS_ASCII.. 27
4.3.4)xdata.. 27
4.3.5)DEBUGENABLED.. 28
4.3.6)Macros to control placement of CRC tables.. 28
4.3.7)INTELLUTION.. 28
4.3.8)INCLUDE_EXCEPTIONS.. 28
4.3.9)DIAGNOSTICS_SUPPORTED.. 29
4.3.10)READ_COILS_SUPPORTED.. 32
4.3.11)WRITE_COILS_SUPPORTED... 32
4.3.12)DISCRETE_INPUTS_SUPPORTED... 32
4.3.13)READ_HOLDING_REGISTERS_SUPPORTED.. 32
4.3.14)WRITE_HOLDING_REGISTERS_SUPPORTED.. 32
4.3.15)READ_WRITE_REGISTERS_SUPPORTED.. 32
4.3.16)MASK_WRITE_REGISTER_SUPPORTED.. 33
4.3.17)GET_PUT_GENERAL_REF_SUPPORTED... 33
4.3.18)FETCH_COMM_EVENT_COUNTER_SUPPORTED.. 33
4.3.19)FETCH_COMM_EVENT_LOG_SUPPORTED.. 33
4.3.20)REPORT_SLAVE_ID_SUPPORTED.. 33
4.3.21)GET_EXCEPTION_COIL_DATA_SUPPORTED... 33

4.3.22)ENCAPSULATED_INTERFACE_TRANSPORT_SUPPORTED........................ 33
4.3.23)READ_FIFO_QUEUE_SUPPORTED.. 34
4.4)Define BYTE and WORD data types and also enumerate TRUE/FALSE......34
5.0)List of Global Variables... 35
6.0)MODBUS Error checking and other information..36
7.0)Calling the MODBUS query message handler 37
a)Poll Mode Calling Method...37
b)Interrupt Mode Calling Method..37
8.0)Tips for optimization the SCL 39
9.0)Technical Specifications.. 40

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.



1.0) The MODBUS Slave Stack Source Code Library

The MODBUS Slave Source Code Library (SCL) is an attempt towards assisting Original Equipment Manufacturers in quickly implementing MODBUS support into their devices. The Slave SCL is an ANSI ‘C’ implementation of the MODBUS Slave 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 Slave 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.

2.0) 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:

2.0.1) 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.

2.0.2) 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.

2.0.3) ‘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.

2.1) 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.

2.1.1) MODBUS Data types

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

2.1.1.1) Coils – These are digital outputs. Coils can be read or written to. A possible value for Coils is either ‘0’ or ‘1’.

2.1.1.2) 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.

2.1.1.3) 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.

2.1.1.4) Input Registers – 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.

2.1.2) 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. Starting Release 1.5 onwards, the MODBUS Slave supports broadcast addressing. On receiving a valid request from a master with a valid slave address, the device replies the Master with an appropriate frame. Starting Release 1.5 onwards the MODBUS Slave Source Code Library supports exception response also.

2.1.3) MODBUS Data Point addressing

MODBUS uses unique addresses to point to data points. Each of the four data types have 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.

3.0) Components of the MODBUS Slave Source Code Library

The MODBUS Slave SCL is implemented using the following files:

3.0.1) MODBUS_Slave.c – This file contains the implementation of the MODBUS Slave Stack. It consists of all functions required to respond to a MODBUS request and all the error handling code. This file also contains the declarations for the global variables created by the stack. The end user should not make modifications to this file.

3.0.2) MODBUS_Slave.h – This file contains header declarations required by the MODBUS_Slave.c implementation. The end user should not make modifications to this file.

3.0.3) MODBUS_User.h – This is the file which 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.

3.0.4) MODBUS_User.c – This file is a sample, basic implementation of the Macro definitions corresponding to the Macros declared in MODBUS_User.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.

4.0) 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 four steps as below:

4.0.1) Define and implement the User Application Interface Macros and Functions

4.0.1.1) GETSLAVEADDR
4.0.1.2) CHECKADDRESSES
4.0.1.3) GETDATA
4.0.1.4) PUTDATA
4.0.1.5) GET_EXCEPTION_COIL_DATA
4.0.1.6) GET_DEVICE_SPECIFIC_DATA
4.0.1.7) GET_GENERAL_REF
4.0.1.8) PUT_GENERAL_REF
4.0.1.9) READ_DEVICE_IDENTIFICATION
4.0.1.10)RESTART_COMMUNICATIONS
4.0.1.11)GET_DIAGNOSTIC_REG_VAL
4.0.1.12)SLAVE_ENTERS_LISTEN_ONLY_MODE
4.0.1.13)CLEAR_COUNTERS_AND_DIAGNOSTIC_REGISTER
4.0.1.14)CLEAR_OVERRUN_COUNTER_AND_FLAG
4.0.1.15)READFIFO

4.0.2) Define and implement the Physical Layer Interface Macros and Functions

4.0.2.1) GETCOMMNPATHNO
4.0.2.2) INITCOMMNPATH
4.0.2.3) FLUSHBUFFER
4.0.2.4) READFROMCOMMNPATH
4.0.2.5) WRITETOCOMMNPATH
4.0.2.6) CLOSECOMMNPATH

4.0.3) Define the Stack Control Macros

4.0.3.1) LITTLE_ENDIAN
4.0.3.2) MODBUS_TCP
4.0.3.3) xdata
4.0.3.4) DEBUGENABLED
4.0.3.5) INTELLUTION
4.0.3.6) INCLUDE_EXCEPTIONS
4.0.3.7) DIAGNOSTICS_SUPPORTED
4.0.3.7.1)RETURN_QUERY_DATA_SUPPORTED
4.0.3.7.2)RESTART_COMM_OPTION_SUPPORTED
4.0.3.7.3)RETURN_DIAGNOSTIC_REG_SUPPORTED
4.0.3.7.4)FORCE_LISTEN_ONLY_MODE_SUPPORTED
4.0.3.7.5)CLEAR_COUNTERS_DIAGNOSTIC_REG_SUPPORTED
4.0.3.7.6)RETURN_BUS_MSG_COUNT_SUPPORTED
4.0.3.7.7)RETURN_BUS_COMM_ERR_SUPPORTED
4.0.3.7.8)RETURN_BUS_EXCPTN_ERR_SUPPORTED
4.0.3.7.9)RETURN_SLAVE_MSG_COUNT_SUPPORTED

4.0.3.7.10)RETURN_SLAVE_NO_RESP_COUNT_SUPPORTED
4.0.3.7.11)RETURN_SLAVE_NAK_COUNT_SUPPORTED
4.0.3.7.12)RETURN_SLAVE_BUSY_COUNT_SUPPORTED
4.0.3.7.13)RETURN_BUS_CHAR_OVERRUN_COUNT_SUPPORTED
4.0.3.7.14)CLEAR_OVERRUN_COUNTER_FLAG_SUPPORTED

4.0.3.8) ENCAPSULATED_INTERFACE_TRANSPORT_SUPPORTED

4.0.3.9) READ_FIFO_QUEUE_SUPPORTED

4.0.4) Define BYTE and WORD data types and enumerate TRUE/FALSE

4.1) The User Application Interface Macros and Functions

These Macros call the stack 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:

4.1.1) GETSLAVEADDR( )

This Macro is called by the stack to determine if a received MODBUS request is intended for it or some other slave. The address value returned by this Macro is compared with the address field in the received MODBUS request. If they match, the execution proceeds, else the request is rejected. The Macro must return a BYTE (unsigned char) value representing the address. How the address value itself is obtained is user implementation specific – it may be hard-coded, may be read from a “DIP Switch” or may be read from a memory location.

Expected return value: BYTE (unsigned char), the Device address
Parameters: None

A sample implementation is as below:

```
// MODBUS_User.h file – actual function name is GetSlaveAddress
#define GETSLAVEADDR()          GetSlaveAddress()

// MODBUS_User.c file - always returns slave addr = 5
unsigned char GetSlaveAddress() { return 0x05;}
```

NOTE: The SCL performs Slave Address check even in MODBUS TCP/IP mode. If this behaviour is not desired, the same must be changed in the Modbus_Slave.c file immediately after the call to GETSLAVEADDR.

4.1.2) CHECKADDRESSES(StartAddress, NoOfRegisters, DataType)

This Macro is called by the stack to check if the register/coil/input addresses specified in the MODBUS request are valid or not (i.e. they exist or not). MODBUS identifies a data point by means of an “address”. Most MODBUS requests contain addresses of the data points from which data is to be returned or to which data must be written to. Before processing a request the stack calls this Macro to validate the existence of data points at all the requested addresses. The function implementation should check all addresses starting from “StartAddress” to “StartAddress+NoOfRegisters” to test if a data point lies at the requested address.

Expected Return Value: integer – TRUE if address(es) are valid, FALSE otherwise
Parameters:

4.1.2.1) *WORD StartAddress - The first address of the address block which must be validated*
4.1.2.2) *WORD NoOfRegisters – The number of following consecutive registers starting from ‘StartAddress’ that must be validated*
4.1.2.3) *int DataType - Indicates the kind of data*
- 01h or 05h or 0Fh = Coil -
02h = Discrete Inputs - 03h or 06h or
10h = Holding Registers - 04h = Input Register

IMP: *If the DataType is 05h or 06h then “NoOfRegisters” must be ignored.*

A sample implementation is as below:

```
// MODBUS_User.h file – actual function name is CheckModbusAddress
#define CHECKADDRESSES(StartAddress, NoOfRegisters, DataType)
        CheckModbusAddress(StartAddress, NoOfRegisters, DataType)
```

```
// MODBUS_User.c file – assumes all addresses between 5 to 15 are valid for all datatypes
int CheckModbusAddress(WORD StartAddress, WORD NoOfRegisters, int DataType){
        switch(DataType){
        case 0x05:  case 0x06       /* Ignore "NoOfRegisters"
                if(StartAddress<0x05 || (StartAddress)>0x0F)        return FALSE;
        default:
                if(StartAddress<0x05 || (StartAddress+NoOfRegisters)>0x0F)
                        return FALSE;                                       }
} // end of function
```

4.1.3) GETDATA(StartAddress, NoOfRegisters, Buffer, DataType)

This is an important Macro called by the stack when it receives a read request. The read request may be for Coils, Discrete Inputs, Input Registers or Holding Registers. This function is the interface from the stack to the Users' Database. The Macro passes to the user the addresses for which data is requested, the kind of data requested and a pre-allocated byte buffer in which the data must be stored.

Expected Return Value: BYTE (unsigned char) – 0: If Successful
Appropriate "MODBUS Exception Code": *If not successful*
If a non-zero value is returned by the implementation of this macro, the SCL will interpret it as an error and as a response sends an "Exception Response" with the exception code equal to the value returned by this macro. ***This feature enables slave implementations to selectively reject requests for data by master by returning a meaningful exception code.***

Parameters:

4.1.3.1) *WORD StartAddress* - Address of the first variable in the requested block of variables*, indexed from '1'*

4.1.3.2) *WORD NoOfRegisters – The number of registers to read from "StartAddress"*

4.1.3.3) *unsigned char *Buffer – A pre allocated buffer into which the requested data must be stored. In case the requested data type is digital, a value of 0x01 must be stored in the buffer for a 'high' variable state and 0x00 must be stored for low state. In case of analog data the Buffer must be used as a 'WORD' buffer so that elements 0 and 1 is used for storing first value, 2 and 3 for second value and so on.*

4.1.3.4) *BYTE DataType – The kind of data requested*
- *01h = Coil*
- *02h = Discrete Inputs*
- *03h = Holding Registers*
- *04h = Input Register*

A sample implementation is as below: For demonstration purpose, data is assumed to be available in two global arrays – 'RefData' holding digital values (coils and discrete inputs) and 'RefDataWord' holding analog values. The GetData function simply copies the Data from these global arrays to the buffer.

```
// MODBUS_User.h file – actual function name is GetData
#define GETDATA(StartAddress, NoOfRegisters, Buffer, DataType) GetData(StartAddress,
                                        NoOfRegisters, Buffer, DataType)

// MODBUS_User.c file – simple demo of GETDATA Macro implementation
BYTE GetData (WORD StartAddress, WORD NoOfRegisters, BYTE *Buffer, BYTE DataType) {
        WORD i, * pWord = (WORD*)Buffer; // buffer is used as a 'WORD' buffer for
                                         // analog variables
        StartAddress -= 1;          since our array is zero indexed decrement address
        NoOfRegisters += StartAddress;              // upper limit of address
        switch( DataType ){
        case 0x01: // Coil - copy data from a global array "RefData" into the Buffer
                for(i=StartAddress; i<NoOfRegisters; i++)
```

```
                Buffer[i-StartAddress]=RefData[i];
                break;
        case 0x02:      // Discrete inputs - same as for coils
                for(i=StartAddress; i<NoOfRegisters; i++)
                Buffer[i-StartAddress]=RefData[i];
                break;
        case 0x03: // Holding Registers - copy data from a global array "RefDataWord"
// into Buffer – note how 'Buffer' has been type cast into WORD pointer
                for(i=StartAddress; i<NoOfRegisters; i++)
                pWord[i-StartAddress]=RefDataWord[i];
                break;
        case 0x04:      // Input Register – same as for Holding Registers
                for(i=StartAddress; i<NoOfRegisters; i++)
                pWord[i-StartAddress]=RefDataWord[i];
                break;
        }
        return MBDEFS_READDATASUCCESS;
} // end of function
```

4.1.4) PUTDATA(StartAddress,NoOfRegisters,Buffer, DataType)

This is another important Macro like 'GETDATA' called by the stack when it receives a write request for Coils or Holding Registers. This function forms the interface for the stack to transfer received data into the Users' Database. This Macro passes to the user the Start address at which the data is to be written from, the number of Coils/Registers to be stored, the kind of data transfer requested and a buffer containing the complete data to be stored.

Expected Return Value: BYTE (unsigned char) —TRUE: If Successful
FALSE: If Data storing fails.

Parameters:

4.1.4.1) *WORD StartAddress - The Starting Address of the variables from which a data write is requested indexed from '1'.*

4.1.4.2) *WORD NoOfRegisters – Number of Registers/Coils to be written, beginning from the specified start address.*

4.1.4.3) *WORD *Buffer – A buffer containing the data to be written to user database. In case the data type is digital (as indicated by the next variable 'DataType'), the buffer contains a value of 0x0001 to indicate a 'high' variable state and 0x0000 for a low state. In case of analog data the buffer contains a two byte value. When multiple numbers of Registers/coils are to be stored the buffer contains the data in successive locations.*

4.1.4.4) *BYTE DataType – The kind of data requested*
- 05h or 0Fh = Coil
- 06h or 10h = Holding Registers

A sample implementation is as below: For demonstration purpose, data is assumed to be made available in two global arrays – 'RefData' for holding digital values (coils) and 'RefDataWord' holding analog values. The PutData function simply copies the data from the buffer to these global arrays.

```
// MODBUS_User.h file – actual function name is PutData
#define PUTDATA(StartAddress, NoOfRegisters , Buffer, DataType)    PutData(StartAddress,
                                        NoOfRegisters , Buffer, DataType)

// MODBUS_User.c file – simple demo of PUTDATA Macro implementation
BYTE PutData(xdata WORD StartAddress, xdata WORD  NoOfRegisters,
            xdata WORD * xdata Buffer, xdata BYTE DataType)
{       StartAddress -= 1;      // convert address to zero indexed to use it on arrays
        NoOfRegisters += StartAddress;
        switch( DataType ){
        case 0x05: case 0x0F: // Coil
```

```
                    for(i=StartAddress; i<NoOfRegisters; i++){
                            RefData[i] = (BYTE)Buffer[i-StartAddress]; } break;
            case 0x06: case 0x10:       // Holding Registers
                    for(i=StartAddress; i<NoOfRegisters; i++){
                            RefDataWord[i] = Buffer[i-StartAddress]; } break;
            }
            return TRUE;
    }// end of function
```

4.1.5) GET_EXCEPTION_COIL_DATA (CoilStsBuffer)

This macro is an interface between the stack and the user by which user provides the eight *Exception* coil status requested by the master from the slave local database. The user must fill the eight exception coil data into the buffer, *CoilStsBuffer*, provided by this macro. The SCL calls this macro when it receives a *Read Exception Status request (FC 0x07)* from a MODBUS Master.

Expected Return Value: Nil
Parameters:

4.1.5.1) unsigned char *CoilStsBuffer – Single Byte where the exception coil status has to be stored.

A sample implementation is as below:

```
// MODBUS_User.h file – actual function name is  GetExceptionCoilData
#define GET_EXCEPTION_COIL_DATA( CoilStsBuffer) GetExceptionCoilData(CoilStsBuffer)

// MODBUS_User.c file – simple demo of  GET_EXCEPTION_COIL_DATA  Macro implementation
void GetExceptionCoilData(unsigned char *CoilStsBuffer){
        /* Get the 8 Exception coil status requested by the master from the slave local database.*/
        *CoilStsBuffer      =TRUE;          /* on  */
        *(CoilStsBuffer + 1)=FALSE;         /* off */
        *(CoilStsBuffer + 2)=TRUE;          /* on  */
        *(CoilStsBuffer + 3)=TRUE;          /* on  */
        *(CoilStsBuffer + 4)=FALSE          /* off */;
        *(CoilStsBuffer + 5)=TRUE;          /* on  */
        *(CoilStsBuffer + 6)=TRUE;          /* on  */
        *(CoilStsBuffer + 7)=FALSE;         /* off */
}// end of function
```

4.1.6) GET_DEVICE_SPECIFIC_DATA (DevSpecsBuf, DataSize)

This macro is an interface between the stack and the user by which user provides the data specific to a type of controller. The user fills the **DevSpecsBuf** with the Slave Id, Run Indicator Status and additional data specfic to that device. The user also must put the total size in bytes of the valid data in **DataSize.** The SCL calls this macro when it receives a *Report Slave ID (FC 0x11)* from a MODBUS Master.

Expected Return Value: Nil
Parameters:

4.1.6.1) unsigned char *DevSpecsBuf – This preallocated buffer passed by the SCL is filled with data specific to the device by the user application.

4.1.6.2) unsigned char *DataSize – Total Number of valid data bytes in the Buffer. Based on the value in this variable the MODBUS frame will be generated by the stack.

A sample implementation is as below:

```
// MODBUS_User.h file – actual function name is   GetDeviceSpecificData
#define GET_DEVICE_SPECIFIC_DATA( DevSpecsBuf,DataSize ) GetDeviceSpecificData
(DevSpecsBuf,DataSize)
```

```
// MODBUS_User.c file – simple demo of   GET_DEVICE_SPECIFIC_DATA Macro implementation
void GetDeviceSpecificData(unsigned char *DevSpecsBuf, unsigned char *DataSize)
{
        /*Get the Slave Id, Run Indicator Status and additional device specific data.
          The data contents are specific to each type of controller. */
        *DevSpecsBuf              =0x08; /* Slave Id */
        *(DevSpecsBuf + 1)        =0xFF; /* RunIndicator Status */
        *(DevSpecsBuf + 2)        =0x01; /* Additional Data */
        *(DevSpecsBuf + 3)        =0x02;
        *(DevSpecsBuf + 4)        =0x03;
        *(DevSpecsBuf + 5)        =0x04;
        *(DevSpecsBuf + 6)        =0x05;

        *DataSize         = 7;    /* Total number of Valid data filled in buffer */
}// end of function
```

4.1.7) GET_GENERAL_REF (FileNumber, StartAddress, RegCount, DataBuf)

The SCL calls this macro when it receives a *Read File Record (FC 0x14 / 0X06)* request from a MODBUS Master. The stack passes as parameters the file number, start address and quantity of registers to the user as requested by the Master device. The user must fill the register data into the buffer **DataBuf** and return to the stack. If successful, this macro returns TRUE and if unsuccessful it returns FALSE, which means the user defined function to this macro should return TRUE, if successful and FALSE, if unsuccessful.

Expected Return Value: BYTE (unsigned char) —TRUE: If Successful
FALSE: If Data storing in buffer fails.

Parameters:

4.1.7.1) WORD FileNumber – File Number
4.1.7.2) WORD StartAdd – Start Address of the File Record.
4.1.7.3) WORD RegCount – Total Number of File Records.
4.1.7.4) WORD *DataBuf – Fill the preallocated buffer passed by the SCL with data in the File Records determined by FileNumber, StartAdd and RegCount.

A sample implementation is as below:

```
// MODBUS_User.h file – actual function name is   GetGeneralRef
#define GET_GENERAL_REF( FileNumber, StartAddress, RegCount,   DataBuf) GetGeneralRef(
FileNumber, StartAddress, RegCount, DataBuf)

// MODBUS_User.c file – simple demo of   GET_GENERAL_REF   Macro implementation
unsigned char GetGeneralRef(WORD FileNumber, WORD StartAdd, WORD RegCount, WORD
*DataBuf)
{
        /* Get the General Referrence Data / File Record */
        *(DataBuf)          =0x0DFE;
        *(DataBuf + 1)      =0X0020;
        return TRUE;
}// end of function
```

4.1.8) PUT_GENERAL_REF (FileNumber, StartAddress, RegCount, DataBuf)

The SCL calls this macro when it receives a *Write File Record (FC 0x15 / 0x06)* request from a MODBUS Master. The stack provides the file number, start address, quantity of registers and a buffer, **DataBuf,** with data to the user as sent by the Master device. If successful, this macro returns TRUE and if unsuccessful, it returns FALSE, which means the user defined function to this macro should return TRUE, if successful and FALSE, if unsuccessful.

Expected Return Value: BYTE (unsigned char) —TRUE: If Successful
FALSE: If Data storing to File Records fails.

Parameters:

4.1.8.1) WORD FileNumber – File Number
4.1.8.2) WORD StartAdd – Start Address of the File Record.
4.1.8.3) WORD RegCount – Total Number of File Records.
4.1.8.4) WORD *DataBuf – Buffer contains data to be written to the File Records determined by FileNumber, StartAdd and RegCount.

A sample implementation is as below:

```
// MODBUS_User.h file – actual function name is  PutGeneralRef
#define  PUT_GENERAL_REF(  FileNumber,  StartAddress,  RegCount,   DataBuf)  PutGeneralRef(
FileNumber, StartAddress, RegCount,  DataBuf)

// MODBUS_User.c file – simple demo of  PUT_GENERAL_REF  Macro implementation
unsigned char PutGeneralRef(WORD  FileNumber,WORD StartAddress, WORD RegCount, WORD
*DataBuf)
{
 /* Update the Registers referenced by FileNumber, StartAddress and  RegCount with the value given in
DataBuf */
-----------------------------------------------
-----------------------------------------------
    -----------------------------------------------
return TRUE;
}// end of function
```

4.1.9) READ_DEVICE_IDENTIFICATION (RDIReadDevId, RDIObjectId, RDIConformityLevel, RDIMoreFollows, RDINextObjId, RDINumOfObjs, RDIObjBuffer, RDIBufferSize)

This macro falls under the *Encapsulated Interface Transport* function of the MODBUS standard. The SCL calls this macro when it receives a *Encapsulated Interface Transport* request from a modbus master with the MEI type set to 0x0E (indicating Read Device Identification request). A MODBUS master uses this request to read device identification data like vendor name, product name etc. from a modbus slave device. An important difference of this request is that it can span multiple transaction since transmitting all device identification information in one modbus frame may not be possible. The user must fill the “device identification” data in the buffer, **RDIObjBuffer**, and total number of bytes filled in the buffer is in the **RDIBufferSize**. The user defined function to this macro should return TRUE, if successful and FALSE, if unsuccessful.

Note: This is a relatively complicated MODBUS function and proper care must be taken in porting it to a device.

The stack passes several other parameters via the macro which must be filled to enable the SCL to make the proper response for a Read Device Identification request. The details of each of the parameters can be found in the MODBUS standard specification document (see section “Technical Specifications”). Parts of it are reproduced here for ready reference.

a) unsigned char RDIReadDevId: This parameter indicates the type of device indentification desired by by the MODBUS Master as given in the request. This parameter is a input (readonly) parameter whose value must only be tested by the user's macro implementation. This parameter can have the following values.

01 : request to get the basic device identification (stream access)
02 : request to get the regular device identification (stream access)
03 : request to get the extended device identification (stream access)
04 : request to get one specific identification object (individual access)

Stream access is one where the master requests for a stream of identification objects which are then sent one after the other. Individual access is one where the master can requests for values of specific objects by indicating their object ID's. All slave devices supporting MEI type 0x0E are required to support stream access whereas the individual access support is optional.

IMP: The macro implementation must return FALSE if *RDIReadDevId* has a value other than the above four.

Values 01 to 03 indicate the level of information that the master desires where as value 04 indicates that the master desires the value of a specific identification object indicated in the parameter **RDIObjectId** (i.e individual access). The information to be supported for each conformity level (basic, regular and extended is as given below).

Basic Device Identification. All objects of this category are mandatory : VendorName, Product code, and revision number.
Regular Device Identification. In addition to Basic data objects, the device provides additional and optional identification and description data objects. All of the objects of this category are defined in the standard but their implementation is optional .
Extended Device Identification. In addition to regular data objects, the device provides additional and optional identification and description private data. All of these data are device dependent.

Object Id	Object Name / Description	Type	M/O	category
0x00	VendorName	ASCII String	**Mandatory**	**Basic**
0x01	ProductCode	ASCII String	**Mandatory**	
0x02	MajorMinorRevision	ASCII String	**Mandatory**	
0x03	VendorUrl	ASCII String	Optional	**Regular**
0x04	ProductName	ASCII String	Optional	
0x05	ModelName	ASCII String	Optional	
0x06	UserApplicationName	ASCII String	Optional	
0x07 … 0x7F	*Reserved*		Optional	**Extended**
0x80 … 0xFF	*Private objects may be **optionally** defined The range [0x80 – 0xFF] is Product dependant.*	device dependant	Optional	

b) unsigned char RDIObjectId: Each information object that can be transacted via MEI type 0x0E is identified by a unique ID called the *Object ID*. The *RDIObjectId* parameter represents this ID and has different meanings in different situations. This parameter is a input (readonly) parameter.
- if *RDIReadDevId = 01, 02 or 03* this parameter indicates the value of the first of a sequence of objects that is being requested by a master. The macro implementation must copy into the passed buffer, identification objects starting with this object ID.
- if *RDIReadDevId=04,* this parameter contains the ID of the information object being requested by the master.

c) unsigned char *RDIConformityLevel: This parameter indicates the conformance of the slave device to the level of information it can provide to a master and also the supported type of access. Depending on the device's design, the user's macro implementation must set this parameter to any of the values below:

0x01 : basic identification (stream access only)
0x02 : regular identification (stream access only)
0x03 : extended identification (stream access only)
0x81 : basic identification (stream access and individual access)
0x82 : regular identification (stream access and individual access)
0x83 : extended identification (stream access and individual access)

d) unsigned char *RDIMoreFollows: This parameter indicates if the slave device has more information objects that can be queried by a master. This is a write only parameter and has no meaning if read by the macro implementation. The valid values for the same must be as follows:

- *In case of RDIReadDevId codes 01, 02 or 03 (stream access),*
As noted above, if the identification data does not fit into a single response, several request/response transactions may be required.
0x00 : no more Objects are availableto be read
0xFF : other identification Objects are available and further MODBUS transactions are required
- *In case of ReadDevId code 04 (individual access),*
this field must be set to 0x00.

e) unsigned char *RDINextObjId: As noted earlier, the information requested by a modbus master device may not be transferable in one MEI response. In such cases, the slave device must use *RDINextObjId* along with *RDIMoreFollows* to indicate to a master if more objects are present to be read. In the macro implementation the following logic must be implemented for supporting requests spanning multiple transactions. When a master makes a first read identification request with type as one of the stream access types (i.e. *RDIReadDevId = 01, 02 or 03)*, it sets *RDIObjectId* to zero (0). The macro, after copying a limited number of information objects into the passed buffer, must set *RDINextObjId* to the next available Object ID to be queried by the master to read the remaining data and also must set *RDIMoreFollows* to 0xFF. The master will then make another *read identification* request with the value of *RDIObjectId* set to object ID value in *RDINextObjId* of previous call. This cycle continues till the slave sets *RDIMoreFollows* to 0x00. If no more objects are present to be read (i.e. *RDIMoreFollows=0x00)*, this parameter must be set to 0x00. The *RDINextObjId* is a output (write only) parameter.

f) unsigned char *RDINumOfObjs: The macro implementation must set the value of this parameter to the total number of identification objects being returned in the passed buffer in this call to the macro.

g) unsigned char *RDIObjBuffer: This is a preallocated buffer passed by the SCL to the macro for storage of the requested identification object data. The following sequence must be followed in copying data to this buffer:

Byte-0 : *Object0.Id* ID of the first Object returned (in case of stream access) or requested Object (in case of individual access)
Byte-1 : *Object0.Length* Length of the first Object in byte
Byte-2 to n : *Object0.Value* Value of the first Object (Object0.Length bytes)
…
Byte-x : *ObjectN.Id* Identification of the last Object (within the response)
Byte-(x+1) : *ObjectN.Length* Length of the last Object in byte
Byte-(x+2) to m: *ObjectN.Value* Value of the last Object (ObjectN.Length bytes)

IMP: Care must be taken to ensure that the total length of the MEI response data does not exceed maximum MODBUS frame lengths. In order to adhere to this limitation, if the number of bytes of data in *RDIObjBuffer* is likely to **exceed 200 bytes**, split the MEI response into multiple fragments by setting *RDIMoreFollows=0xFF.*

h) unsigned char *RDIBufferSize: The macro implementation must set this parameter to the number of bytes of data (including the Object.Id and Object.Length fields) copied into *RDIObjBuffer.*

Expected Return Value: BYTE (unsigned char) —TRUE: If Successful
FALSE: If Data storing to File Records fails.

A sample implementation is as below:

```
// MODBUS_User.h file – actual function name is   ReadDeviceIdentification
#define READ_DEVICE_IDENTIFICATION (RDIReadDevId, RDIObjectId, RDIConformityLevel,
RDIMoreFollows, RDINextObjId, RDINumOfObjs, RDIObjBuffer, RDIBufferSize)
ReadDeviceIdentification (RDIReadDevId, RDIObjectId, RDIConformityLevel,RDIMoreFollows,
RDINextObjId, RDINumOfObjs, RDIObjBuffer, RDIBufferSize)

// MODBUS_User.c file – simple demo of   READ_DEVICE_IDENTIFICATION   Macro implementation
unsigned char ReadDeviceIdentification(unsigned char RDIReadDevId, unsigned char
      RDIObjectId, unsigned char *RDIConformityLevel,unsigned char *RDIMoreFollows,       unsigned
char *RDINextObjId, unsigned char *RDINumOfObjs, unsigned char                     *RDIObjBuffer,
unsigned char *RDIBufferSize)
{
      switch (RDIReadDevId)
      {
      case 0x01: /* request  to get the basic device identification (stream access) */
              {
   char companyid[]={"Company Identification"};
                      char productcode[]={"Product Code"};
                      char version[]={"V2.11"};
                       *RDIConformityLevel        =        0x01;
                       *RDIMoreFollows            =        0x00;
                       *RDINextObjId              =        0x00;
                       *RDINumOfObjs              =        0x03;
                       *RDIObjBuffer              =        0x00;    /* Object Id */
                       *(RDIObjBuffer+1)          =        0x16;    /* Object Length */
memcpy((RDIObjBuffer+2),& companyid ,0x16);
                       *(RDIObjBuffer+24)         =        0x01;  /* Object Id */
                       *(RDIObjBuffer+25)         =        0x0C;  /* Object Length */
                       memcpy((RDIObjBuffer+26),&productcode,0x0C);
                       *(RDIObjBuffer+26+0x0C)=            02;      /* Object Id */
                       *(RDIObjBuffer+27+0x0C)=            05;      /* Object Length */
                       memcpy((RDIObjBuffer+26+0x0C+2),&version,0x05);
               *RDIBufferSize = 26+0x0C+0x02+0x05; /* Total Valid filled Bufferdata */
              }
              break;
      case 0x02: /* request to get the regular device identification (stream access) */
              {
--------------------------------------
              }
              break;
      case 0x03: /* request to get the  extended device identification (stream access) */
              {
--------------------------------------
              }
              break;
      case 0x04: /* request to get one specific identification object (individual access) */
              {
--------------------------------------
              }
              break;
      default:   /* Error */
              {
                      return FALSE;
              }
      }
      return TRUE;
}// end of function
```

4.1.10) RESTART_COMMUNICATIONS ( )

The SCL calls this macro to performs the necessary actions to be taken to restart the communication hardware of the device. This macro gets called as a part of *diagnostic function*

request (FC 0x08, sub-function code = 0x01) of MODBUS. This macro implementation must restart the communication hardware of the slave device, reset all communication counters and initialise the communication port. This macro should return TRUE, if successful and FALSE, if unsuccessful.

Expected Return Value: BYTE (unsigned char) —TRUE: If Successful
FALSE: If Data storing to File Records fails.

Parameters:None

A sample implementation is as below:

```
// MODBUS_User.h file – actual function name is  RestartCommunications
#define RESTART_COMMUNICATIONS()          RestartCommunications()

// MODBUS_User.c file – simple demo of RESTART_COMMUNICATIONS Macro implementation
unsigned char RestartCommunications()
{
	/*	The actions to be taken for a restart command.
		Mainly used under Diagnostics --Restart Communications Option */
   ------------------------------------------------
   ------------------------------------------------
   ------------------------------------------------
	return TRUE;
}// end of function
```

4.1.11) GET_DIAGNOSTIC_REG_VAL (DiagRegVal)

This macro is called when the SCL receives a *diagnostic function request (FC 0x08)* with *sub-function code = 0x02*. The user's macro implementation must return the *diagnostic register* value in the slave by copying the value of the same into the *DiagRegVal* parameter.

Expected Return Value:Nil

Parameters:

4.1.11.1) unsigned short * DiagRegVal – Two Byte Diagnostic Register Value has to be stored in this preallocated short variable passed by this SCL.

A sample implementation is as below:

```
// MODBUS_User.h file – actual function name is   GetDiagnosticRegVal
#define GET_DIAGNOSTIC_REG_VAL(DiagRegVal) GetDiagnosticRegVal(DiagRegVal)

// MODBUS_User.c file – simple demo of   GET_DIAGNOSTIC_REG_VAL Macro implementation
void GetDiagnosticRegVal(unsigned short * DiagRegVal)
{
	/* Gets the 16 bit diagnostic register value */
	/* Mainly used under Diagnostics             */
	*DiagRegVal=0xF0F0;      /* 16 bit Diagnostic Register Value */
}// end of function
```

4.1.12) SLAVE_ENTERS_LISTEN_ONLY_MODE ( )

This macro is called when the SCL receives a *diagnostic function request (FC 0x08)* with *sub-function code = 0x04*. Once this mode is entered, the SCL only receives requests but does not respond to the same till it is forced out of the listen only mode. This macro is provided as an interface to the user's application where-in necessary actions at the application level can be taken to reflect the device's new mode. This macro should return TRUE, if successful and FALSE, if unsuccessful.

Expected Return Value: BYTE (unsigned char) —TRUE: If Successful
FALSE: If Data storing to File Records fails.
Parameters:None

A sample implementation is as below:

```
// MODBUS_User.h file – actual function name is   SlaveEntersListenOnlyMode
#define SLAVE_ENTERS_LISTEN_ONLY_MODE() SlaveEntersListenOnlyMode()

// MODBUS_User.c file – simple demo of SLAVE_ENTERS_LISTEN_ONLY_MODE Macro
implementation
unsigned char SlaveEntersListenOnlyMode()
{
        /*The actions to be taken for a Listen Only Mode command. */
  /* Mainly used under Diagnostics.*/
 ----------------------------------
 ----------------------------------
        return TRUE;
}// end of function
```

4.1.13) CLEAR_COUNTERS_AND_DIAGNOSTIC_REGISTER ( )

The SCL calls this macro on receiving a *diagnostic function request (FC 0x08)* with *sub-function code = 0x0A.* The macro implementation must clear the internal counters and diagnostic register of the slave device. This macro should return TRUE, if successful and FALSE, if unsuccessful.

Expected Return Value: BYTE (unsigned char) —TRUE: If Successful
FALSE: If Data storing to File Records fails.
Parameters:None

A sample implementation is as below:

```
// MODBUS_User.h file – actual function name is   ClearCountersAndDiagnosticRegister
#define CLEAR_COUNTERS_AND_DIAGNOSTIC_REGISTER()
ClearCountersAndDiagnosticRegister()

// MODBUS_User.c file – simple demo of   CLEAR_COUNTERS_AND_DIAGNOSTIC_REGISTER
Macro implementation
unsigned char ClearCountersAndDiagnosticRegister()
{
        /* Clear the Counters and Diagnostic Register. Mainly used under Diagnostics.*/
  ----------------------------------
  ----------------------------------
        return TRUE;
}// end of function
```

4.1.14) CLEAR_OVERRUN_COUNTER_AND_FLAG ( )

The SCL calls this macro on receiving a *diagnostic function request (FC 0x08)* with *sub-function code = 0x14.* This macro implementation must clear the device's overrun error counter and reset the error flag in the slave device. This macro should return TRUE, if successful and FALSE, if unsuccessful.

Expected Return Value: BYTE (unsigned char) —TRUE: If Successful
FALSE: If Data storing to File Records fails.
Parameters:None
A sample implementation is as below:

```
// MODBUS_User.h file – actual function name is   ClearOverrunCounterAndFlag
#define CLEAR_OVERRUN_COUNTER_AND_FLAG()      ClearOverrunCounterAndFlag()

// MODBUS_User.c file – simple demo of   CLEAR_OVERRUN_COUNTER_AND_FLAG   Macro
implementation
unsigned char ClearOverrunCounterAndFlag()
{
        /* Clear the Overrun Counter and Flags. Mainly used under Diagnostics.*/
        ---------------------------------
   ---------------------------------
        ---------------------------------
        return TRUE;
}// end of function
```

4.1.15) READFIFO (FIFOPntrAdd, FIFOCount, FIFORegBuf)

This macro reads the contents of a First-In-First-Out (FIFO) queue of register in a slave device. It is called against a *Read FIFO Queue (FC 0x18)* request from a master. This macro is an interface between user and stack that provides the user with the FIFO pointer address. The user defined function should fill the FIFO buffer data in *FIFORegBuf* and total number of FIFO data filled in the *FIFORegBuf* in the *FIFOCount*. If successful, this macro must return TRUE and if unsuccessful FALSE.

Expected Return Value: BYTE (unsigned char) —TRUE: If Successful
FALSE: If Data storing to File Records fails.

Parameters:

4.1.15.1) WORD *FIFOPntrAdd - Address of the FIFO Pointer

4.1.15.2) WORD *FIFOCount - Total Number of FIFO Bytes should be put here. It should be less than or equal to 31.

4.1.15.3) WORD *FIFORegBuf - Fill this buffer with FIFO data.

A sample implementation is as below:

```
// MODBUS_User.h file – actual function name is   ReadFifo
#define READFIFO(FIFOPntrAdd, FIFOCount, FIFORegBuf) ReadFifo(FIFOPntrAdd, FIFOCount,
FIFORegBuf)

// MODBUS_User.c file – simple demo of   READFIFO  Macro implementation
unsigned char ReadFifo(WORD *FIFOPntrAdd, WORD *FIFOCount, WORD *FIFORegBuf)
{
*FIFORegBuf  =          0x01B8;
*(FIFORegBuf + 1) =     0x1284;
    *FIFOCount          =         2;                /* 2 items in FIFO Queue */
     return TRUE;
}// end of function
```

4.2) 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:

4.2.1) GETCOMMNPATHNO()

This Macro must be called by the user application during the application startup to enable the stack to setup 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 can be done either in this Macro itself or in the 'INITCOMMNPATH' Macro. A sample implementation for a Windows based application can be as below:
Expected Return Value: DWORD PathID – a unique identifier to the comm. path

Parameters: None

A typical implementation for Windows would be as below:

```
// MODBUS_User.h file
#define GETCOMMNPATHNO() OpenCommPort()

// MODBUS_User.c file – a Windows implementation for a serial communication port
DWORD OpenCommPort(){ HANDLE hCom; // open “COM1” port and return Handle
        hCom = CreateFile(“COM1”, GENERIC_READ | GENERIC_WRITE,0, NULL,
        OPEN_EXISTING,  0, NULL );
        return (DWORD)hCom;         }
```

4.2.2) INITCOMMNPATH(CommnPathNo)

Similar to 'GETCOMMNPATHNO' this macro must be called by the user application during the application startup to enable the stack to initialize its communication path. The implementation of this Macro should contain the necessary code to initialize and setup the communication path decided by the 'GETCOMMNPATHNO' macro. For e.g. for proper communication, the baud rate, parity, data bits etc. must be setup for a typical serial communication path. Implementation of this macro is not mandatory if the initialization is already done in the 'GETCOMMNPATHNO' macro. The unique communication path identifier returned by 'GETCOMMNPATHNO' is passed as a parameter to this macro to enable initialization of the right port.
Expected Return Value: TRUE if successful FALSE otherwise

Parameters:
DWORD CommnPathNo – The unique communication path identifier returned by the GETCOMMNPATHNO macro.

A typical implementation for Windows would be as below:

```
// MODBUS_User.h file
#define INITCOMMNPATHNO(CommnPathNo) InitCommPort(CommnPathNo)

// MODBUS_User.c file – a Windows implementation for a serial communication port
int InitCommPort(DWORD CommnPathNo){
        HANDLE hCom = (HANDLE) CommnPathNo; DCB dcb;
```

```
        GetCommState(hCom, &dcb); // get existing port parameters
        /* Fill in the DCB: baud=9600, 8 data bits, no parity, 1 stop bit. */
        dcb.BaudRate = 9600; dcb.ByteSize = 8;
        dcb.Parity = NOPARITY;   dcb.StopBits = ONESTOPBIT;
        SetCommState(hCom, &dcb); // set up port with newly configured parameters
        Return TRUE;
}
```

4.2.3) READFROMCOMMNPATH(CommnPathNo, nBytes, Buffer)

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 and should return the number of bytes of data read. If the requested number of bytes cannot be read, then the function should return with the actual number of bytes read. If an error is encountered in the reading process, the function should return with a negative value indicating failure. Since the execution of the stack *blocks* till this function returns, care must be taken in its implementation to build in a “timeout” processing logic so that the function returns if no data is available at the communication path after the specified timeout.

Expected Return Value:

int – If Success: The number of bytes actually read from the communication path.

If Failure : A negative value if an error is encountered in reading from the communication path.

- *value = -1 : Error reading communication path*
- *value = -2 : Timeout occurred*

Parameters:

DWORD CommnPathNo – A unique ID returned by the GETCOMMNPATHNO representing the communication path

int nBytes – Number of bytes to read from path

*unsigned char *Buffer – A pre-allocated buffer to store received bytes*

IMP: The SCL does not implement any timeout within the stack. Hence this macro implementation must ensure proper timeout processing so that the stack does not lockup inside the read macro call when no data is received on the communication port.

A typical Windows implementation is as below:

```
// MODBUS_User.h file
#define READFROMCOMMNPATH(CommnPathNo, nBytes, Buffer)
                                    ReadSerialPort(CommnPathNo, nBytes, Buffer)

// MODBUS_User.c file – a Windows implementation for a serial communication port with
//no timeout processing
int ReadSerialPort(DWORD CommnPathNo, int nBytes, unsigned char *Buffer){
        int NoOfBytesWritten;       int i;
        if (!ReadFile( (HANDLE)CommnPathNo, /* handle of file to read */
                                    Buffer, /* address of buffer that receives data */
                                    nBytes,/* number of bytes to read */
                                    &NoOfBytesWritten,/* address of number of bytes read */
                                    NULL /* address of structure for data */
    )) return -1;
        else return NoOfBytesWritten;
} // end of function
```

4.2.4) WRITETOCOMMNPATH(CommnPathNo, nBytes, Buffer)

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 return the number of bytes of data written. If the requested number of bytes cannot be written, then the function should return with the actual number of bytes written. If an error is encountered in the writing process, the function should return with a negative value indicating failure.

Expected Return Value:

int – If Success: The number of bytes actually written from the communication path.
If Failure : A negative value if an error is encountered in reading from the communication path.

- *value = -1 : Error reading communication path*
- *value = -2 : Timeout occurred*

Parameters:
DWORD CommnPathNo – A unique ID returned by the GETCOMMNPATHNO representing the communication path
int nBytes – Number of bytes to write to path
*unsigned char *Buffer – A pre-allocated buffer containing the data to be written to the port.*

IMP: The SCL does not implement any timeout within the stack. Hence this macro implementation must ensure proper timeout processing so that the stack does not lockup inside the write macro call.

A typical Windows implementation is as below:

```
// MODBUS_User.h file
#define WRITETOCOMMNPATH(CommnPathNo, nBytes, Buffer)
                        WriteSerialPort(CommnPathNo, nBytes, Buffer)
// MODBUS_User.c file – a Windows implementation for a serial communication port
int WriteSerialPort(DWORD CommnPathNo, int nBytes, unsigned char *Buffer){
        int NoOfBytesWritten;
        if (!WriteFile( (HANDLE)CommnPathNo, /* handle to file to write to */
                        Buffer,/* pointer to data to write to file */
                        nBytes, /* number of bytes to write */
                        &NoOfBytesWritten, /* pointer to number of bytes written */
                        NULL /* pointer to structure needed for overlapped I/O */
                )) return -1;
        else return NoOfBytesWritten;
}
```

4.2.5) FLUSHBUFFER(CommnPathNo)

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 the communication path of all data.

Expected Return Value:
TRUE if successful,
FALSE otherwise.

Parameters:
DWORD CommnPathNo – A unique ID returned by the GETCOMMNPATHNO representing the communication path whose buffer is to be cleared.

A typical Windows implementation is as below:

```
// MODBUS_User.h file
#define FLUSHBUFFER(CommnPathNo) FlushBuffer(CommnPathNo)

// MODBUS_User.c file – a Windows implementation for a serial communication port
int FlushBuffer(DWORD CommnPathNo){
return PurgeComm((HANDLE) CommnPathNo
```

```
PURGE_RXCLEAR| PURGE_TXCLEAR); }
```

4.2.6) CLOSECOMMNPATH(CommnPathNo)

The user application must call 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 stacks/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

Parameters:

DWORD CommnPathNo – A unique ID returned by the GETCOMMNPATHNO representing the communication path which is to be closed.

A typical Windows implementation is as below:

```
// MODBUS_User.h file
#define CLOSECOMMNPATH(CommnPathNo) CloseCommPort(CommnPathNo)

//MODBUS_User.c file
int CloseCommPort(DWORD CommnPathNo){ return CloseHandle( (HANDLE) CommnPathNo); }
```

4.3) 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:

4.3.1) 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 as 1.

e.g.

```
#define LITTLE_ENDIAN   1        /* Store in Little Endian format */
#define LITTLE_ENDIAN   0        /* Store in Big Endian format */
```

4.3.2) MODBUS_TCP

This symbol must be defined as 1, if the MODBUS stack is being used for communication over TCP/IP and 0, if if the MODBUS stack is being used for communication over Serial line. 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 and TCP/IP standards 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.

```
#define MODBUS_TCP    1  /* MODBUS over TCP/IP is used */
#define MODBUS_TCP    0  /* MODBUS over Serial  is used */
```

4.3.3) MODBUS_ASCII

MODBUS supports two types of communication modes, MODBUS RTU and MODBUS ASCII. In order to select the communication mode as RTU, set this macro as 0 and to select the communication mode as ASCII, set this macro as 1. ASCII mode support is available for serial communication only, hence set the macro MODBUS_TCP as 0 if MODBUS_ASCII is set to1.

4.3.4) xdata

The Source Code Library uses several global and local variables. On some platforms all variables are created on the internal memory of the CPU by default. Since the internal memory of most CPUs is limited, there must be a way of explicitly instructing the compiler to create the variables in “external memory”. In the source code library this is achieved by means of the ‘xdata’ directive. Since the keyword to create data in external memory changes from platform to platform, the symbol ‘xdata’ must be redefined to the keyword supported by the platform being used. For e.g. if the keyword to create a variable in external memory is EXT_MEM then xdata must be redefined as:

```
#define xdata EXT_MEM
```

If the platform under use creates all variables in external memory by default, then ‘xdata’ must be redefined as:

```
#define xdata
```

4.3.5) 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 as 1, else define it as 0.

eg:-

```
#define DEBUGENABLED 1      /* Debug Statements were enabled */
#define DEBUGENABLED 0      /* Debug Statements were disabled */
```

4.3.6) Macros to control placement of CRC tables

The MODBUS RTU standard uses a CRC creation logic that requires two tables of 256 bytes each containing a few prefixed values. The total memory required to hold these tables is 512 bytes which is a significant amount of memory considering that many micro controllers have just below 1KB of RAM. In order to optimize memory usage the SCL provides a means for specifying if the CRC tables are to be maintained statically and where OR if the values in the table are to be dynamically calculated. This is done by means of setting the values of the following macros:

- CRC_TABLE_IN_RAM
- CRC_TABLE_IN_ROM
- CRC_TABLE_CREATE_DYNAMIC

In a given configuration of the SCL, only one of the above three macros can be set to one (1). The others should be set to zero(0).

If CRC_TABLE_IN_RAM is set to 1, then the CRC tables are place in RAM thus consuming 512 bytes of RAM. This results in fastest code execution but uses up the most memory.

If CRC_TABLE_IN_ROM is set to 1 then the CRC tables are created as constant arrays in ROM/Flash (please check if your compiler creates "const" variables in ROM). This uses up more code ROM and results in slightly lesser speed than the above option but saves precious RAM.

If CRC_TABLE_CREATE_DYNAMIC is set to '1', then the values of the CRC tables are created dynamically every time the CRC is generated. This is the most expensive option in terms of speed as the CRC tables get dynamically created for every frame received or sent but is the least expensive in terms of RAM and ROM usage (it uses only 4 bytes of RAM as against the other two options.

4.3.7) INTELLUTION

This macro is now obsolete. Always set to macro to zero (default setting in the delivered SCL).

4.3.8) INCLUDE_EXCEPTIONS

If the support for Exception Response is required at the MODBUS side, set this macro to 1, otherwise set to 0. If set to 0, the stack does not send any response if an exception condition occurs in processing requests from master.

eg:-

```
#define INCLUDE_EXCEPTIONS  1 /* Exception Response support enabled*/
#define INCLUDE_EXCEPTIONS  0 /* Exception Response support disabled */
```

Note: The following macros allow fine tuning of the Source Code Library to selectively enable support for different MODBUS functions. This permits optimizing the size of the final binary and also the memory consumption by excluding source code for MODBUS functions that are not intended to be supported.

4.3.9) DIAGNOSTICS_SUPPORTED

If Diagnostic functions are supported by slave device, this macro should be set to 1, otherwise set to 0. If this macro is set to 1, the below defined fourteen macros should be set to either 0 or 1, depending on the sub functions supported by the slave device. As per the MODBUS standard, this support is available for Serial Line MODBUS only.

eg:-
```
#define DIAGNOSTICS_SUPPORTED  1  /* Diagnostic support enabled */
#define DIAGNOSTICS_SUPPORTED  0  /* Diagnostic support disabled */
```

4.3.9.1) RETURN_QUERY_DATA_SUPPORTED

If this macro is set to 1, whenever the Master device request this function code, the response frame similar to the request is send back to the Master. If set to 0, an exception response is sent back to the Master, if INCLUDE_EXCEPTIONS is set to 1, else ignored.

eg:-
```
#define RETURN_QUERY_DATA_SUPPORTED  1  /* Return Query Data
                                 Diagnostic support enabled */
#define RETURN_QUERY_DATA_SUPPORTED  0  /* Return Query Data
                                 Diagnostic support disabled */
```

4.3.9.2) RESTART_COMM_OPTION_SUPPORTED

If this macro is set to 1, whenever the Master device requests, the slave device's serial line port will be restarted and initialized, and all of its communication event counters may be cleared. If the slave device supports '**Listen Only Mode**', then this macro should be set to 1 in order to bring the slave out of 'Listen Only Mode'. Otherwise set to 0.

eg:-
```
#define RESTART_COMM_OPTION_SUPPORTED  1  /* Restart
                         Communication support enabled */
 #define RESTART_COMM_OPTION_SUPPORTED  0  /* Restart
                         Communication support disabled */
```

4.3.9.3) RETURN_DIAGNOSTIC_REG_SUPPORTED

If the slave device has a diagnostic register, then set this macro to 1. Otherwise set to 0.

eg:-
```
#define RETURN_DIAGNOSTIC_REG_SUPPORTED  1  /* Return
                Diagnostic Register value support enabled */
#define RETURN_DIAGNOSTIC_REG_SUPPORTED  0  /* Return
                Diagnostic Register value support disabled */
```

4.3.9.4) FORCE_LISTEN_ONLY_MODE_SUPPORTED

If the slave device supports '**Listen Only Mode**', set this macro to 1. Otherwise set to 0. In order to bring slave out of '**Listen Only Mode**', set the macro **RESTART_COMM_OPTION_SUPPORTED** also to 1.

eg:-
```
#define FORCE_LISTEN_ONLY_MODE_SUPPORTED  1  /* Slave
                           Supports Listen Only mode */
#define FORCE_LISTEN_ONLY_MODE_SUPPORTED  0  /* Slave don't
                           Supports Listen Only mode */
```

4.3.9.5) CLEAR_COUNTERS_DIAGNOSTIC_REG_SUPPORTED

If this macro is set to 1, the Master can request to clear the counters and diagnostic register present in the slave device. Otherwise set to 0.

eg:-
```
#define  CLEAR_COUNTERS_DIAGNOSTIC_REG_SUPPORTED  1
        /* Clear the counters and diagnostic registers available in the slave */
#define  CLEAR_COUNTERS_DIAGNOSTIC_REG_SUPPORTED  0
/* Donot Clear the counters and diagnostic registers available in the slave */
```

4.3.9.6) RETURN_BUS_MSG_COUNT_SUPPORTED

If this macro is set to 1, the total bus message count will be returned since the device's last restart, clear counters operation, or power–up, if the Master device requests the same. Otherwise set to 0.

eg:-
```
#define  RETURN_BUS_MSG_COUNT_SUPPORTED  1
        /* Stack will return Bus Message count on master request */
#define  RETURN_BUS_MSG_COUNT_SUPPORTED  0
          /* Stack will return Exception Response on Master Request for Bus Message
count */
```

4.3.9.7) RETURN_BUS_COMM_ERR_SUPPORTED

If this macro is set to 1, the total CRC error count will be returned since the device's last restart, clear counters operation, or power–up, if the Master device requests the same. Otherwise set to 0.

eg:-
```
#define RETURN_BUS_COMM_ERR_SUPPORTED     1
        /* Stack will return CRC Error count on master request */
#define  RETURN_BUS_COMM_ERR_SUPPORTED     0
          /* Stack will return Exception Response on Master Request for CRC Error
count */
```

4.3.9.8) RETURN_BUS_EXCPTN_ERR_SUPPORTED

If this macro is set to 1, the count of exception error responses sent to the Master device will be returned since the device's last restart, clear counters operation, or power–up, if the Master device requests the same. Otherwise set to 0.

eg:-
```
#define  RETURN_BUS_EXCPTN_ERR_SUPPORTED  1
          /* Stack will return Bus Exception Error count on Master request */
#define  RETURN_BUS_EXCPTN_ERR_SUPPORTED  0
          /* Stack will return Exception response on Master request for Bus Exception
Error count*/
```

4.3.9.9) RETURN_SLAVE_MSG_COUNT_SUPPORTED

If this macro is set to 1, the total count of messages addressed to the remote device, or broadcast, that the remote device has processed since its last restart, clear counters operation, or power–up, is sent to the Master device when requested. Otherwise set to 0.

eg:-
```
#define  RETURN_SLAVE_MSG_COUNT_SUPPORTED  1
          /* Slave Message count will be returned on Master request */
#define  RETURN_SLAVE_MSG_COUNT_SUPPORTED  0
            /* Stack will return Exception response on Master request for Slave
            Message count*/
```

4.3.9.10) RETURN_SLAVE_NO_RESP_COUNT_SUPPORTED

If this macro is set to 1, the quantity of messages addressed to the remote device for which it has returned no response (neither a normal response nor an exception response), since its last restart, clear counters operation, or power–up, is sent to the Master device if the Master device requests the same. Otherwise set to 0.

```
eg:-
#define RETURN_SLAVE_NO_RESP_COUNT_SUPPORTED 1
        /* Slave No Response count will be returned on Master request */
#define RETURN_SLAVE_NO_RESP_COUNT_SUPPORTED 0
        /* Stack will return Exception response on Master request for Slave No
        Response count*/
```

4.3.9.11) RETURN_SLAVE_NAK_COUNT_SUPPORTED

If this macro is set to 1, the quantity of messages addressed to the remote device for which it returned a Negative Acknowledge (NAK) exception response, since its last restart, clear counters operation, or power–up, is sent to the Master device if the Master device requests the same. Otherwise set to 0.

```
eg:-
#define RETURN_SLAVE_NAK_COUNT_SUPPORTED 1
        /* Slave NAK count will be returned on Master request */
#define RETURN_SLAVE_NAK_COUNT_SUPPORTED 0
        /* Stack will return Exception response on Master request for Slave NAK
        count*/
```

4.3.9.12) RETURN_SLAVE_BUSY_COUNT_SUPPORTED

If this macro is set to 1, the quantity of messages addressed to the remote device for which it returned a Slave Device Busy exception response, since its last restart, clear counters operation, or power–up, is sent to the Master device if the Master device requests the same. Otherwise set to 0.

```
eg:-
#define RETURN_SLAVE_BUSY_COUNT_SUPPORTED 1
        /* Slave Busy count will be returned on Master request */
#define RETURN_SLAVE_BUSY_COUNT_SUPPORTED 0
        /* Stack will return Exception response on Master request for Slave Busy
        count*/
```

4.3.9.13) RETURN_BUS_CHAR_OVERRUN_COUNT_SUPPORTED

If this macro is set to 1, the quantity of messages addressed to the remote device that it could not handle due to a character overrun condition, since its last restart, clear counters operation, or power–up, is sent to the Master device if the Master device requests the same. Otherwise set to 0.

```
eg:-
#define RETURN_BUS_CHAR_OVERRUN_COUNT_SUPPORTED 1
/* Slave Bus character overrun count will be returned on Master request */
#define RETURN_BUS_CHAR_OVERRUN_COUNT_SUPPORTED 0
/* Stack will return Exception response on Master request for Slave character overrun
count*/
```

4.3.9.14) CLEAR_OVERRUN_COUNTER_FLAG_SUPPORTED

If this macro is set to 1, the Master device can request the slave to clear the overrun error counter and reset the error flag in the slave device. Otherwise set to 0.

```
eg:-
#define CLEAR_OVERRUN_COUNTER_FLAG_SUPPORTED 1
/* Slave clears the overrun counter and clears the eror flag */
#define CLEAR_OVERRUN_COUNTER_FLAG_SUPPORTED 0
/* Stack will return Exception response on Master request */
```

4.3.10) READ_COILS_SUPPORTED

Set this macro to one (1) for enabling support for MODBUS Function Code 0x01 (Read Coils) else set to zero (0). If set to zero, the library responds to a request with FC 0x01 with an *ILLEGAL FUNCTION (code 0x01)* exception response.

4.3.11) WRITE_COILS_SUPPORTED

Set this macro to one (1) for enabling support for MODBUS Function Codes 0x05 and 0x0F (Write Single Coil and Write Multiple Coils respectively) else set to zero (0). If set to zero, the library responds to a request with FC 0x05 or FC 0x0F with an *ILLEGAL FUNCTION (code 0x01)* exception response.

4.3.12) DISCRETE_INPUTS_SUPPORTED

Set this macro to one (1) for enabling support for MODBUS Function Code 0x02 (Read Discrete Inputs) else set to zero (0). If set to zero, the library responds to a request with FC 0x02 with an *ILLEGAL FUNCTION (code 0x01)* exception response.

4.3.13) READ_HOLDING_REGISTERS_SUPPORTED

Set this macro to one (1) for enabling support for MODBUS Function Code 0x03 (Read Holding Registers) else set to zero (0). If set to zero, the library responds to a request with FC 0x03 with an *ILLEGAL FUNCTION (code 0x01)* exception response. Additionally this macro together with the macros WRITE_HOLDING_REGISTERS_SUPPORTED and READ_WRITE_REGISTERS_SUPPORTED controls if MODBUS Function Code 0x17 (Read Write Multiple Registers) is supported. If both the READ_HOLDING_REGISTERS_SUPPORTED macro and the WRITE_HOLDING_REGISTERS_SUPPORTED macros are set to one (1) then the FC 0x17 is supported if READ_WRITE_REGISTERS_SUPPORTED macro is one (1). If either one is zero then the library responds to a FC 0x17 request with an *ILLEGAL FUNCTION (code 0x01)* exception response irrespective of the value of READ_WRITE_REGISTERS_SUPPORTED.

4.3.14) WRITE_HOLDING_REGISTERS_SUPPORTED

Set this macro to one (1) for enabling support for MODBUS Function Codes 0x06 and 0x10 (Write Single Register and Write Multiple Registers respectively) else set to zero (0). If set to zero, the library responds to a request with FC 0x06 or FC 0x10 with an *ILLEGAL FUNCTION (code 0x01)* exception response. Besides controlling support for FC 0x06 and 0x10, this function indirectly affects two other function codes also.

- Together with the macros READ_HOLDING_REGISTERS_SUPPORTED and READ_WRITE_REGISTERS_SUPPORTED this macro controls if MODBUS Function Code 0x17 (Read Write Multiple Registers) is supported. If both the READ_HOLDING_REGISTERS_SUPPORTED macro and the WRITE_HOLDING_REGISTERS_SUPPORTED macros are set to one (1) then FC 0x17 is supported if READ_WRITE_REGISTERS_SUPPORTED macro also is set to one (1). If either of the two is set to zero then the library responds to a FC 0x17 request with an *ILLEGAL FUNCTION (code 0x01)* exception response irrespective of the value of READ_WRITE_REGISTERS_SUPPORTED.
- Function code 0x16 (Mask Write Register) is supported if the macro MASK_WRITE_REGISTER_SUPPORTED is set to one (1) if and only if the WRITE_HOLDING_REGISTERS_SUPPORTED macro is also set to one (1).

4.3.15) READ_WRITE_REGISTERS_SUPPORTED

Set this macro to one (1) for enabling support for MODBUS Function Code 0x17 (Read Write Multiple Registers) else set to zero (0). If set to zero, the library responds to a request with FC 0x17 with an *ILLEGAL FUNCTION (code 0x01)* exception response. However the effectiveness of this macro is dependent on two other macros - READ_HOLDING_REGISTERS_SUPPORTED and WRITE_HOLDING_REGISTERS_SUPPORTED. Setting

READ_WRITE_REGISTERS_SUPPORTED to one (1) will provide FC 0x17 support only if both the READ_HOLDING_REGISTERS_SUPPORTED macro and the WRITE_HOLDING_REGISTERS_SUPPORTED macros are also set to one (1). If either one is zero then the library responds to a FC 0x17 request with an *ILLEGAL FUNCTION (code 0x01)* exception response irrespective of the value of READ_WRITE_REGISTERS_SUPPORTED.

4.3.16) MASK_WRITE_REGISTER_SUPPORTED

Set this macro to one (1) for enabling support for MODBUS Function Code 0x16 (Mask Write Register) else set to zero (0). If set to zero, the library responds to a request with FC 0x16 with an *ILLEGAL FUNCTION (code 0x01)* exception response. However the effectiveness of this macro is dependent on the macro WRITE_HOLDING_REGISTERS_SUPPORTED. If this macro is set to zero (0) then FC 0x16 is not supported irrespective of the value of MASK_WRITE_REGISTER_SUPPORTED.

4.3.17) GET_PUT_GENERAL_REF_SUPPORTED

Set this macro to one (1) for enabling support for MODBUS Function Codes 0x14 and 0x15 (Read General Reference/File Record and Write General Reference/File Record respectively) else set to zero (0). If set to zero, the library responds to a request with FC 0x14 or FC 0x15 with an *ILLEGAL FUNCTION (code 0x01)* exception response.

4.3.18) FETCH_COMM_EVENT_COUNTER_SUPPORTED

Set this macro to one (1) for enabling support for MODBUS Function Code 0x0B (Fetch Comm Event Counter) else set to zero (0). If set to zero, the library responds to a request with FC 0x0B with an *ILLEGAL FUNCTION (code 0x01)* exception response. This macro is effective in MODBUS RTU mode only (i.e. when MODBUS_TCP macro is set to zero).

4.3.19) FETCH_COMM_EVENT_LOG_SUPPORTED

Set this macro to one (1) for enabling support for MODBUS Function Code 0x0C (Fetch Comm Event Log) else set to zero (0). If set to zero, the library responds to a request with FC 0x0C with an *ILLEGAL FUNCTION (code 0x01)* exception response. This macro is effective in MODBUS RTU mode only (i.e. when MODBUS_TCP macro is set to zero).

4.3.20) REPORT_SLAVE_ID_SUPPORTED

Set this macro to one (1) for enabling support for MODBUS Function Code 0x11 (Report Slave ID) else set to zero (0). If set to zero, the library responds to a request with FC 0x11 with an *ILLEGAL FUNCTION (code 0x01)* exception response. This macro is effective in MODBUS RTU mode only (i.e. when MODBUS_TCP macro is set to zero).

4.3.21) GET_EXCEPTION_COIL_DATA_SUPPORTED

Set this macro to one (1) for enabling support for MODBUS Function Code 0x07 (Read Exception Status) else set to zero (0). If set to zero, the library responds to a request with FC 0x07 with an *ILLEGAL FUNCTION (code 0x01)* exception response. This macro is effective in MODBUS RTU mode only (i.e. when MODBUS_TCP macro is set to zero).

4.3.22) ENCAPSULATED_INTERFACE_TRANSPORT_SUPPORTED

The MODBUS Encapsulated Interface (MEI) Transport is a mechanism for tunneling service requests and method invocations, as well as their returns, inside MODBUS PDUs. If this macro is set to 1, MEI Transport mechanism is supported by the slave device, Otherwise set to 0.

eg:-

```
#define ENCAPSULATED_INTERFACE_TRANSPORT_SUPPORTED 1
                        /* Encapsulated Interface Transport support enabled */
#define ENCAPSULATED_INTERFACE_TRANSPORT_SUPPORTED 0
                        /* Encapsulated Interface Transport support disabled */
```

4.3.23) READ_FIFO_QUEUE_SUPPORTED

The function code 24 (0x18) allows reading the contents of a First-In-First-Out (FIFO) queue of register in a remote device. The function returns a count of the registers in the queue, followed by the queued data. Up to 32 registers can be read: the count, plus up to 31 queued data registers. If this macro is set to 1, indicates that the slave device supports Read FIFO QUEUE Function Code =24 (0x18). Otherwise set to 0.

```
eg:-
#define  READ_FIFO_QUEUE_SUPPORTED  1
                                          /* Read FIFO support enabled */
#define  READ_FIFO_QUEUE_SUPPORTED  0
                                          /* Read FIFO support disabled */
```

4.4) Define BYTE and WORD data types and also enumerate TRUE/FALSE

Most non-windows targets do not define the data type WORD and BYTE. To ensure clarity, the SCL uses BYTE for single byte data types and WORD for two byte data types. On platforms which do not define these data types, the same must be made here explicitly. Similarly TRUE and FALSE must be enumerated. Proper care must be taken when defining these data types since the working of the SCL is crucial to these definitions. Specifically the size of ‘int’ differs from platform to platform – anywhere between two bytes to four bytes. So the right keyword whose size is two bytes must be chosen for defining WORD. A demonstration of this is given below. It is assumed that on this platform ‘int’ is a two byte value.

```
// WORD is an unsigned int – occupies two bytes of memory
   typedef unsigned int WORD;

// BYTE is an unsigned char – occupies one byte of memory
typedef unsigned char BYTE;

// enumerate FALSE and TRUE: FALSE = 0, TRUE=1
   enum {FALSE,TRUE};
```

5.0) List of Global Variables

The stack uses many 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 'MBSlave_'. These variables can be found in the 'Global Variable Declarations' section of the MODBUS_Slave.c file. The same has been repeated here for ease of use. 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) BYTE MBSlave_BitCount - counter to track bit number when packing digital status
2) BYTE MBSlave_SetBit - indicates a digital high - used to pack digital status
3) int MBSlave_nLeft - represents the number of bytes left to be read from serial buffer
4) int MBSlave_nRead - represents the number of bytes actually read by a read request
5) int MBSlave_FunctionCode - MODBUS function code found in request frame
6) int MBSlave_nCoils - number of coil data requested in read/write request frame
7) int MBSlave_nHoldingRegs - number of holding registers requested in the read/write request frame
8) int MBSlave_IsValidStartAddr - flag to track if a requested address is valid as per address pool or not
9) BYTE MBSlave_ValidMessage – flag to indicate that received MODBUS request is valid
10) QUERY_MESG_HEADER MesgHeader - From modbus.h file : All query messages contain this standard header
11) unsigned long MBSlave_ValidMesgCnt - Valid Query messages received
12) unsigned long MBSlave_TotalMesgCnt - Total Query messages received till now
13) WORD MBSlave_StartAddress - place holder for storing start address of the of the requested data block
14) WORD MBSlave_LastMbErr - Last modbus 'Query message' receive error variable
15) BYTE MBSlave_SendBuffer[CurSendBufSize] - A buffer for Response message to MODBUS Master
16) BYTE MBSlave_InitOK - CRC Calculation related
17) BYTE MBSlave_Table1[256] - CRC Calculation related
18) BYTE MBSlave_Table2[256] - CRC Calculation related
19) unsigned long gMBSlave_CRCErrCount - Bus Communication Error Count
20) unsigned long gMBSlave_BusExcptnErrCount - Bus Exception Error Count
21) unsigned long gMBSlave_NoRespCount - Slave No Response Count
22) unsigned long gMBSlave_NAKCount - Slave NAK Count
23) unsigned long gMBSlave_BusyCount - Slave Busy Count
24) unsigned long gMBSlave_BusCharOverrunCount - Bus Character Overrun Count
25) unsigned char gCommEventLogBuf[64] - Circular Array – Event Log Buffer
26) unsigned char gCommEventLogBufIndex - Index for the gCommEventLogBuf - varies from 0 to 63, then rolls back
27) unsigned char gTotalValidEventLogEntries - Total number of Valid Event Log Entries - varies from 0 to 64
28) unsigned char CmdProcessGoingOn - Indicates Slave busy or not
29) unsigned char SlaveInListenOnlyMode – Indicates Slave is in Listen Only Mode or not

6.0) MODBUS Error checking and other information

The SCL stores error information on the error it encounters in a global variable named 'MBSlave_LastMbErr' of type WORD. 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 MODBUS_Slave.h file and are reproduced below for easy reference.

1.1) EINVALFUNCTCODE - function code in the query message is other than those supported
1.2) EINVALSTARTADDR - query message contains an address which is not valid
1.3) EINVALDBREQ - query message contains request for out of range data bytes
1.4) EINVALREADVALUE - the digital value to be read or written is not 00 or 01
1.5) ETIMEOUT - Timeout occurred reading MODBUS frame
1.6) EBADCRC - Frame contained a Bad CRC
1.7) EBUFOVERRUN - Frame too large to contain in the buffer
1.8) EINVALSLAVEADDR - Slave address in frame does not match our address
1.9) ECOMMPATHERROR - Error occurred reading commn. Path

Two more global variables that are of interest to the user application are:

a) *unsigned long MBSlave_ValidMesgCnt* – This indicates the total number of valid query messages received by the stack since it started. This value is continuously incremented whenever valid query messages are received and can be read by the user application at any time.
b) *unsigned long MBSlave_TotalMesgCnt* – This variable indicates the total number of query messages received by the stack including those which were invalid. This variable is incremented every time the stack receives a MODBUS query before it does a validity check on the frame.

7.0) Calling the MODBUS query message handler

The main interface function from the user application to the SCL is the 'HandleModbusRequest' interface function. This function must be passed a valid communication path ID obtained by a previous call to the GETCOMMNPATHNO macro and duly initialized by the INITCOMMNPATH macro. This function attempts to read a MODBUS query message from the communication path by calling the READFROMCOMMNPATH macro, parses the received frame and if required sends a reply by calling the WRITETOCOMMNPATH macro. If it requires data from the user database it calls the GETDATA macro and calls PUTDATA to write data to the user database. This function returns TRUE (unsigned char, value=1) when the MODBUS query is handled successfully. It returns FALSE(unsigned char, value=0) if it encounters a CRC error or if the READFROMCOMMNPATH macro returns error. In such a case the global variable "MBSlave_LastMbErr" should be checked to see the type of error occurred and accordingly necessary action can be taken. This function can be called in two modes:

a) Poll Mode Calling Method

In this mode this function is called by the user application cyclically in a loop irrespective of availability of any data in the communication path buffer. In this mode this function tries to read data from the communication path and processes the received data. If there was no data in the communication path buffer, the function times out and returns an error. An example of such a call is as given below:

```
void main(){
        BYTE QUIT = 0, bRetval;
        WORD CommnPathNo;      /* A path ID to the communication channel. */
        CommnPathNo = GETCOMMNPATHNO();  /* Get path no. on which to
                                                  communicate */
         INITCOMMNPATH(CommnPathNo);  /* initialise the communication channel */
        /* Do receive, parse and send continuously till asked to quit */
        QUIT = FALSE;
        while(!QUIT){
         DoUserApplSpecificTasks();   /* first do all user application specific tasks */
         bRetval =HandleModbusRequest(CommnPathNo); /* see if a MODBUS query was
                                    received and process  it if found */
        CallErrorHandlingFunction(MBSlave_LastMbErr); /*Check for the type of error
                                             occurred and take necessary action*/
        }
} /* end of main */
```

b) Interrupt Mode Calling Method

In this mode the user application sets up an interrupt handler to be triggered whenever data arrives on the communication channel and the 'HandleModbusRequest' function is called/invoked from this interrupt handler. This way the function is invoked only when there is data on the communication channel which leads to a more efficient usage of the system resources. In the main function the user application continues to execute as usual. An example will look as below:

```
/* Interrupt based usage of the HandleModbusRequest function – method - 1*/
WORD CommnPathNo;     /* Global Variable - A path ID to the communication channel */
int SerialISR()
{
  /* Data received on communication channel – process it */
   HandleModbusRequest(CommnPathNo);
}
```

```
void main(){
   BYTE QUIT = 0;

  CommnPathNo = GETCOMMNPATHNO();  /* Get path no. on which to communicate */
  INITCOMMNPATH(CommnPathNo);       /* initialise the communication channel */
  SetUpISR(SerialISR);
   /* Do user specific tasks continuously till asked to quit */
   QUIT = FALSE;
   while(!QUIT){
          DoUserApplSpecificTasks();   /* first do all user application specific tasks */
   }
} /* end of main */
```

There may be situations where an interrupt service routine or handler cannot afford to execute a large function like HandleModbusRequest. In such cases an alternate implementation as below can be used.

```
/* Interrupt based usage of the HandleModbusRequest function – method - 2*/
int DataReceived = 0;
int SerialISR()
{
          DataReceived = 1; /* Data received – Set appropriate flag */
}
void main(){
  BYTE QUIT = 0;
  WORD CommnPathNo;    /* Global Variable - A path ID to the communication channel */
  CommnPathNo = GETCOMMNPATHNO();  /* Get path no. on which to communicate */
  INITCOMMNPATH(CommnPathNo);       /* initialise the communication channel */
  SetUpISR(SerialISR);
   /* Do user tasks continuously till quit – process MODBUS messages if data in path */
   QUIT = FALSE;
   while(!QUIT){
          DoUserApplSpecificTasks();   /* first do all user application specific tasks */
          /* If data was received on the comm. Path process it */
          if(DataReceived){ DataReceived = 0; HandleModbusRequest(CommnPathNo); }
   }
} /* end of main */
```

The above implementation ensures that the HandleModbusRequest function will not be called till data is received in the communication buffer.

8.0) Tips for optimization the SCL

It is important to carefully set the values for various configuration macros in order to optimize the use of your systems resources by the source code library. Following are some of the settings that can help minimize the use of resource by the stack:

1. Exclude unsupported MODBUS functions by setting the corresponding xxx_SUPPORTED macro to zero (0). This reduces code size to a great extent by excluding the source code of the unsupported functions from compilation.
2. If your target is short of RAM then it is a good idea to configure the CRC placement macros to either place the CRC tables in ROM or to get the CRC table values to be calculated dynamically every time the CRC is to be evaluated. Be aware that the second option increases CPU load as the CRC table values get calculated for every frame being sent or received.
3. If you are severely short of memory or code ROM to fit the SCL please contact Sunlux Technologies by email at support@sunlux-india.com. There are a few more optimizations possible which require some changes in the SCL core implementation and is best left for the design team to handle. Please include details of RAM and ROM availability on your device as well as the MODBUS configuration of your device (number of coils/discrete inputs/holding registers/input registers).

9.0) Technical Specifications

Parameter	Value
Standard	MODBUS Application Protocol Specification V1.1a, June 2004, 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	• 01 (0x01) Read Coils
	• 02 (0x02) Read Discrete Inputs
	• 03 (0x03) Read Holding Registers
	• 04 (0x04) Read Input Registers
	• 05 (0x05) Write Single Coil
	• 06 (0x06) Write Single Register
	• 07 (0x07) Read Exception Status (Serial Line only)
	• 08 (0x08) Diagnostics (Serial Line only)
	    • 00 (0x00) Return Query Data
	    • 01 (0x01) Restart Communications Option
	    • 02 (0x02) Return Diagnostic Register
	    • 04 (0x04) Force Listen Only Mode
	    • 10 (0x0A) Clear Counters and Diagnostic Register
	    • 11 (0x0B) Return Bus Message Count
	    • 12 (0x0C) Return Bus Communication Error Count
	    • 13 (0x0D) Return Bus Exception Error Count
	    • 14 (0x0E) Return Slave Message Count
	    • 15 (0x0F) Return Slave No Response Count
	    • 16 (0x10) Return Slave NAK Count
	    • 17 (0x11) Return Slave Busy Count
	    • 18 (0x12) Return Bus Character Overrun Count
	• 11 (0x0B) Get Comm Event Counter (Serial Line only)
	• 12 (0x0C) Get Comm Event Log (Serial Line only)
	• 15 (0x0F) Write Multiple Coils
	• 16 (0x10) Write Multiple registers
	• 17 (0x11) Report Slave ID (Serial Line only)
	• 20 / 6 (0x14 / 0X06) Read File Record
	• 21 / 6 (0x15 / 0x06) Write File Record
	• 22 (0x16) Mask Write Register
	• 23 (0x17) Read/Write Multiple registers
	• 24 (0x18) Read FIFO Queue
	• 43 (0x2B) Encapsulated Interface Transport
	    • 14 (0x0E) Read Device Identification

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 ‘ANSI C’ supporting platform