MODBUS Slave Source Code Library

Getting Started
And Implementation Guide

Version 2.0

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

Introduction

The Getting Started and Implementation Guide is a document that will help you in commencing the porting of the MODBUS Slave Source Code Library to your native platform. Your package of the MODBUS Slave Source Code Library contains the following items:

1) The MODBUS Slave Source Code Library 'C' Source Files at the location <CD Drive>:\Source Code Library
2) Sample Win32 port of the SCL at the location <CD Drive>:\Ports\Win32
3) A MODBUS Master test application to validate your implementation at the location <CD Drive>:\Test Application\ModbusTest.exe
4) A Test Procedure Document to test your implementation of the SCL at the location <CD Drive>:\Documents\ MBSlaveSCL-TP.pdf
5) The MODBUS Slave Source Code Library Users Manual at the location <CD Drive>:\Documents\ MBSlaveSCL-UM.pdf
6) This document at the location <CD Drive>:\Documents\MBSlaveSCL-IG.pdf

Before starting with the porting of the SCL, please make a backup of all the files in case you have to fallback on the original files. The following sections describe the procedure to be followed in porting the SCL to your platform.

Prerequisites

Before starting with the port, please ensure that:

a) You have a 'C' compiler/development IDE ready for compiling the SCL
b) Brush up your MODBUS basics by going through the relevant section in the Users Guide
c) Get your hardware ready with the selected communication port

Terminologies and Concepts Used

1) **Platform**: Refers to the hardware/software environment that the end user intends to run the SCL on. This may be an OS based platform (e.g. a Windows based Personal Computer) or a bare platform without an operating system (e.g. a Single Board Computer based on MCU 8051).
2) **Porting SCL**: The activity of implementing the user definable MACROS so that the SCL can be executed on the native platform of the user.
3) **Declaration (Prototype) of a function:** is the function prototype that shows the type of the function with the details about the arguments and the return value. For e.g.
   int Add(int Var1, int Var2);
   is an example of the declaration of the function. Observe that the above statement is not the function itself.
4) **Definition (implementation) of a function:** is the actual body of the function. For e.g.
   int Add(int Var1, int Var2){ return Var1+ar2; }

is the implementation of the function 'Add'.

5) **User Definable MACROS:** The SCL is a 'C' language implementation of the MODBUS RTU communication protocol. Since it is not targeted at specific platforms but intended to be used on a variety of platforms, certain sections of the source code is left unimplemented. This unimplemented sections are to be implemented by the end user for the platform that he is using. These unimplemented sections have been written as 'C' MACROS. A 'C' MACRO is a symbolic entry in the source file that can be redefined to a new value. For e.g.
   #define SIZE 10
   'SIZE' above is a simple MACRO that has been defined a '10'. All instances of 'SIZE' are replaced with '10' in the source file containing this MACRO. Going a step ahead, the following is a more complex MACRO that takes arguments:
   #define FUNCTION(Num1, Num2) Add(Num1, Num2)
   'FUNCTION' above is a MACRO that has been defined as a function named 'Add' taking in two parameters. All instances of 'FUNCTION' will be replaced with 'Add' and the arguments passed to 'FUNCTION' will be passed to 'Add' in the same sequence. For e.g. FUNCTION(10,12)
   Will be replaced with Add(10,12)
   Some important points to note:
   - The MACRO definition should ensure that the exact number of parameters is used in the function as is in the MACRO.
     For e.g. #define FUNCTION(Var1) Add(Var1, Var2)
     is invalid.
   - Unlike in case of the simple MACRO like 'SIZE', a MACRO if defined as a function (as was done for FUNCTION above) will have to be accompanied by the definition (i.e. implementation) of the function also. In the above case of the implementation of 'Add' is not included in the source file, the compiler flags an error stating 'Unknown Variable/Function)
   - It is evident that when the MACRO is defined as a function with parameters, there is no information provided on the *type* of the parameters (int, float etc.) nor is any information provided on the return value of the function. This information will rather be available from the definition (implementation) of the function.
6) **Real-time Database:** Refers to the storage/data structure in which the device on which the SCL is ported stores the data acquired by it. The Real-time Database can be a simple global 'C' structure or may be a common 'Shared Memory' pool. The SCL provides some MACROS through which it accesses the Real-time Database.

Planning an Implementation Strategy

The MODBUS SCL has been written so that it is highly flexible and user friendly to use. Due to this reason there can be several implementation schemes possible. It is useful to spend sometime to analyse your system and requirements and arrive at a well-defined strategy to port the SCL. Answers to the following questions should be obtained before starting the implementation:

a) What is the Physical Port on which communication is desirable? It could be asynchronous serial RS232 interface, an RS485 interface, Ethernet, Fiber Optic etc.
b) What is the type if MODBUS implementation desired? MODBUS Serial or MODBUS TCP?
c) For the selected interface above, do you have a device driver? If your platform has an operating system it is likely that the operating system vendor has supplied the device driver. For bare platforms, an explicit driver may not be available in which case the transmission and reception of data/bytes on the physical media is achieved by means of directly accessing the hardware. The porting engineer must get sufficient information on using the physical interface selected.
d) Have your data structures been clearly defined? For e.g. the existing data acquisition program on your hardware may be acquiring field data and storing them in a 'C' structure. The porting engineer must be clear of such data structure since the SCL must be linked/mapped to these data structures.
e) Are you planning a multitasking architecture for running the MODBUS communication driver or is it a monolithic single task implementation? If multitasking is desired then the methodology for sharing data with the MODBUS SCL and the "Data Acquisition" program should be decided. For e.g., a simple 'Shared Memory' can be used to store common data. Another implementation may use 'pipes'. In case of multitasking environment, the porting engineer must be conversant with process synchronization and inter process communication techniques for the native platform.
f) Do you want to provide MODBUS support simultaneously to multiple MODBUS Masters? In such a case multiple instances of the ported SCL must be running as concurrent processes each handling a different Master. This question become relevant in case of MODBUS TCP since multiple Master can communicate with a MODBUS Slave on the same Ethernet interface.
g) Have you decided your MODBUS Register Mapping? MODBUS identifies different data points uniquely by means of register addresses. Every register/coil/digital input in MODBUS has a four digit hexadecimal address. When a MODBUS Master requests data, it sends the addresses of those registers whose data is desired. These addresses must clearly

be mapped to the real-time database of your device and must be documented so that MODBUS Masters know which address to request for which data.

h) How do you intend to configure/set the MODBUS Slave ID for your device? MODBUS being a single-Master multi-Slave protocol, the Master should be able to address a specific MODBUS Slave device. To achieve this, the MODBUS standard provides for a Slave ID, which is a single byte value from 1 to 247, which is unique for a network. The end user must decide on a scheme to provide the Slave Device ID to the SCL when requested for. Typical schemes used to set/change the MODBUS Slave ID of a device is by means of DIP switches. Alternatively, the MODBUS Slave device ID can be hard-coded into the device. However this reduces the flexibility of the device as a MODBUS Slave.

i) Is your device BIG ENDIAN or LITTLE ENDIAN? This refers to the manner in which a processor stores 'WORD' in memory – Hi Byte first and Low Byte next or vice versa. Generally all Intel CPU's follow LITTLE ENDIAN style and all Motorola CPU's follow BIG ENDIAN style.

Implementation Procedure

If you have answers to fairly a good number of questions above, you may start with the implementation now. Follow the procedure described in the steps below to complete the porting of the SCL.

a) Understanding the Source Files

The SCL contains mainly four important files:

i. MODBUS_Slave.c
ii. MODBUS_Slave.h
iii. MODBUS_User.h
iv. MODBUS_User.c

The first two files contain the implementation of the MODBUS protocol and in all probabilities 'may' not be required to be changed by the end user. Most of the porting activities will have to be done in the third and the fourth files. It is suggested that the engineer examine the files MODBUS_User.c and MODBUS_User.h with respect to the explanation in the User Manual and understand the same.

b) Define the User-definable MACROS in MODBUS_User.h file

The SCL contains several MACROS that must be defined by the end user. For a list of MACROS see the Users Manual. The supplied MODBUS_User.h file has all the MACROS predefined. The end user may retain these MACRO definitions or may change it if he feels that the function names mapped to the MACROS are not appropriate to his application. For e.g. supplied SCL file contains a definition for the MACRO **READFROMCOMMNPATH** as 'ReadCommnPath'. Now suppose the end

user is uses a serial RS232 port. He may want to define this MACRO as 'ReadFromSerialPort'. Suppose he uses Ethernet/TCP communication – he may want to define it as 'ReadFromTCPSocket'. As such there is no limitation/restrictions on defining the MACRO provided that the definition complies with the ANSI 'C' standard.

c) Do Skeleton Implementation of the defined MACROS

After defining all the MACROS in the MODBUS_User.h file the next step is to do a skeleton implementation of all the functions defined under the MACROS in the MODBUS_User.h file. The skeleton implementation of the functions does nothing – it simply contains the function header, opening and closing braces. While doing the skeleton implementation care must be taken to match the parameters types and the return values of the functions. The user comments in the MODBUS_User.h or the User Guide may be referred for details of the parameters types and the return values.

d) Add function prototypes (declarations) into MODBUS_User.h

The main SCL file, MODBUS_Slave.c, makes calls to all the user definable MACROS to obtain access to the User Real-time Database and the physical layer of the platform used. Since the definition of these MACROS and the definition of the corresponding functions are present in different files, it is important to make available, at least, the function prototypes of the defined functions to the MODBUS_Slave.c file. Since the MODBUS_User.h file is #included into the MODBUS_Slave.c, this is an appropriate file to write the function prototypes. Please note that the files supplied with the SCL already have these function prototypes included.

e) Begin with implementation of User Application Interface MACROS

The list of MACROS to be defined and the detailed explanation of each of the MACROS can be found in the Users Manual. Specifically the following points must be considered when porting these MACROS:

1) The PUTDATA MACRO has special meaning for the returned values (non zero for success and zero for failure). Such conventions should be strictly followed for the SCL to function properly.
2) The 'StartAddress' parameter passed to the CHECKADDRESSES MACRO is 'one-based' and not 'zero-indexed'. Hence a validity check for StartAddress == 0 must be done without fail. ***Additionally, if the 'DataType' variable has a value 0x05 or 0x06 then the 'NoOfRegisters' parameter must be ignored.***
3) The same MACRO call is made in case of GETDATA and PUTDATA for all MODBUS types. The 'DataType' variable must be tested to determine the kind of variable for which the data is requested.
4) In case of GETDATA MACRO, the 'Buffer' parameter is a BYTE buffer and has the following meanings:
   - 'DataType' is a Register: The 'Buffer' parameter must be interpreted as a WORD by saving 'Buffer' into a WORD pointer. The WORD pointer must be used as an array with each element representing a WORD. See the statement below for a clear understanding.

WORD *pWord = (WORD*)Buffer;
pWord[0]=0x12AB;

The stored WORD should not be adjusted in GETDATA for the BIG/LITTLE ENDIAN difference since this is done internally done in the SCL. Specifically, if the chosen platform is LITTLE ENDIAN, the Low Byte should be stored first and then the Hi Byte and vice versa for BIG ENDIAN.

- 'DataType' is a Coil/Digital Input: Each element of 'Buffer' is a BYTE and the status of each digital variable requested must be saved into one each element of 'Buffer'.
  Buffer[0] = 0x00; // status of one digital variable
  Buffer[1] = 0x01; // storing a 'TRUE' value

5) In case of PUTDATA MACRO, the 'Buffer' parameter is a WORD buffer and has the following meanings:
   - 'DataType' is a Register: The 'Buffer' parameter being a WORD array can be used directly as an array with each element representing a WORD. **LocalWordArray[0] = Buffer[0];** The WORD value contained in 'Buffer' would already be adjusted in the SCL for the BIG/LITTLE ENDIAN difference and as such no further adjustments must be done on the same in PUTDATA.
   - 'DataType' is a Coil/Digital Input: Each element of 'Buffer' is a WORD containing the digital status of one digital variable. See below for usage.
     // status of one digital variable
     LocalByteBuffer[0] = (BYTE)Buffer[0];

f) Proceed with implementation of Physcial Layer Interface MACROS

The list of MACROS to be defined and the detailed explanation of each of the MACROS can be found in the Users Manual. Specifically the following points must be considered when porting these MACROS:

1) Do not forget to implement/set 'timeouts' for the READFROMCOMMNPATH and WRITETOCOMMNPATH MACRO implementation. It this is not done, the call to the HandleModbusRequest will lockup if there is no data in the communication buffer!
2) Both READFROMCOMMNPATH and WRITETOCOMMNPATH MACRO implementations must return the number of successful bytes read/written to the path.
3) The CommnPathNo parameter is a WORD parameter and must be type cast to the actual 'type' of the handle to your communication interface. For e.g. in case of Win32, if you use TCP/IP sockets then

the CommnPathNo variable is of type SOCKET and must be type case before being used in send and receive functions. e.g. recv((SOCKET)CommnPathNo, Buffer, nBytes, 0)

4) Please see the documented return values for the above functions and return the proper value before exiting the functions.
5) GETCOMMNPATHNO has been provided for systems having multiple communication ports. The user implementation may contain necessary code to search for an unused port and return the same.
6) INITCOMMNPATH is an additional MACRO for users who do not want to do communication port initialization in GETCOMMNPATHNO
7) The FLUSHBUFFER MACRO has been provided to clear the communication buffers which may be having residual data from previous communication sessions
8) The use of the MACROS GETCOMMNPATHNO and INITCOMMNPATH is not mandatory. The user may decide to do all communication port opening and initialization within his application and only pass a valid handle (CommnPathNo) to the HandleModbusRequest function.

g) Proceed with the implementation of Stack Control MACROS

Detailed explanation of the stack control MACROS is available in the Users Manual. Some considerations:

1) DEBUGENABLED is a very useful MACRO which, if set to '1', will cause diagnostic messages to be printed on the standard output of your device. These messages can be very useful in debugging your port of the SCL. However for those users which platforms do not support standard output functions or which do not have a standard output, this MACRO must be disabled. For users who do have a low level standard output (character based LCD displays), it is suggested that the users write a basic implementation of 'printf' so as to make use of the diagnostic messages.
2) 'xdata' is another useful MACRO for users who use microcontroller-based targets. Generally simple microcontrollers like MCU 8051, 80C535 etc. have very little on-chip memory space. By default the compilers used for these devices create all variables in the internal RAM of the MCU. This will lead to a memory/stack overrun. One of the most common non-ANSI method of explicitly getting these compilers to create the variables in external memory is by the use of the keyword 'xdata'. Placing xdata in front of variable names results in the variable being created in external memory space. Users who do not wish to use this feature may set xdata to NULL (#define xdata)
3) The MODBUS standard uses the BIG_ENDIAN style of interpreting WORDS and hence on platforms using the BIG ENDIAN style no byte adjustment is required. The LITTLE_ENDIAN MACRO if defined as TRUE will perform a byte reversal of all WORDS that are used in the

SCL. As already noted in some of the above sections, the end users implementation must NEVER do any byte adjustments external to the SCL.

h) *Make calls to GETCOMMNPATHNO, INITCOMMNPATH in the main user program if required*
As already noted, using GETCOMMNPATHNO and INITCOMMNPATH is not mandatory.

i) *Make calls to the main entry function HandleModbusRequest in the main user program in a loop*
Every call to HandleModbusRequest will cause the SCL to try and read the communication path/buffer, test if the received data is a MODBUS request and respond to it accordingly. Calling HandleModbusRequest at a low frequency can cause the device's communication buffer to overflow and thus lose some MODBUS requests due to data corruption. Alternatively, as demonstrated in the Users Manual, the HandleModbusRequest function can be called in interrupt mode as well.

j) Insert the SCL files into your project in the Integrated Development Environment. In addition to the main four files, the SCL uses one more file by name MBDefs.h All these files must be inserted into your IDE in addition to the end user's main program source file. Proceed with the compilation of your project now to create your end application. The SCL source files can be found on the supplied CDROM at the location <CD Drive>:\Source Code Library

You have now completed the porting of the MODBUS Slave Source Code Library. You may now proceed to test your implementation using the MODBUS Master Test application provided with the SCL.