emModbus

CPU independent Modbus
stack for embedded applications

User Guide & Reference Manual

Document: UM14001
Software Version: 1.02f
Revision: 1
Date: September 24, 2018

A product of SEGGER Microcontroller GmbH

www.segger.com

Disclaimer

Copyright notice

Trademarks

Contact address

SEGGER Microcontroller GmbH

Ecolab-Allee 5
D-40789 Monheim am Rhein

Germany

Tel.	+49 2173-99312-0
Fax.	+49 2173-99312-28
E-mail:	support@segger.com
Internet:	www.segger.com

Manual versions

This manual describes the current software version. If any error occurs, inform us and we will try to assist you as soon as possible. Contact us for further information on topics or routines not yet specified.

Print date: September 24, 2018

Software	Revision	Date	By	Description
1.02f	1	180924	OO	Contact details updated.
1.02f	0	180522	OO	Chapter "emModbus API" updated. • MB_SLAVE_ConfigIgnoreSlaveAddr() added.
1.02c	0	170810	OO	Chapter "emModbus API" updated. • Error code MB_ERR_WOULD_BLOCK added. Chapter "Configuring emModbus" • MB_ALLOW_STREAM_HDR_UNDERFLOW added. • MB_DISCONNECT_ON_MSG_TOO_BIG added.
1.02	0	160401	OO	Chapter "emModbus API" updated. • MB_LoadU16BE() added. • MB_StoreU16BE() added. • MB_SLAVE_SetCustomFunctionCodeHandler() added.
1.00b	3	150806	MC/ OO	Minor corrections and layout updates. Chapter "Introduction to emModbus" • Added information about emModbus data handling.
1.00	2	140601	MC	Updated file information.
1.00	1	140314	MC	Chapters "Getting Started" and "Tasks and Interrupt usage" updated. Spelling.
1.00	0	140224	MC	Initial version.

About this document

Assumptions

This document assumes that you already have a solid knowledge of the following:

- The software tools used for building your application (assembler, linker, C compiler).
- The C programming language.
- The target processor.
- DOS command line.

If you feel that your knowledge of C is not sufficient, we recommend *The C Programming Language* by Kernighan and Richie (ISBN 0--13--1103628), which describes the standard in C programming and, in newer editions, also covers the ANSI C standard.

How to use this manual

This manual explains all the functions and macros that the product offers. It assumes you have a working knowledge of the C language. Knowledge of assembly programming is not required.

Typographic conventions for syntax

This manual uses the following typographic conventions:

Style	Used for
Body	Body text.
`Keyword`	Text that you enter at the command prompt or that appears on the display (that is system functions, file- or pathnames).
`Parameter`	Parameters in API functions.
`Sample`	Sample code in program examples.
`Sample comment`	Comments in program examples.
Reference	Reference to chapters, sections, tables and figures or other documents.
GUIElement	Buttons, dialog boxes, menu names, menu commands.
Emphasis	Very important sections.

Table of contents

Chapter 1

Introduction to emModbus

This chapter provides an introduction to emModbus. It explains the basic concept behind emModbus and its modules.

1.1 The Modbus standard

The Modbus protocol was originally published in 1979 by Modicon (which later became Schneider Electric) and has since evolved into a standard communications protocol for industrial electronic devices. In 2004, Schneider transfered rights to the protocol to the Modbus Organization, who now controls the open standard's further development.

1.1.1 Modbus message basics

The Modbus protocol is an application layer messaging protocol used for communications between devices that are connected to different types of buses or networks.

It uses a master-slave-technique in which one device, the master, initiates transactions (called "queries"). Other devices, the slaves, respond by performing the action requested in the query or by supplying the requested data to the master.

The protocol determines how each device will know its address, how it will recognize a message addressed to it, how it will determine the kind of action to be taken and how it will extract data or any other information contained in the message. It also determines how slaves construct and send reply messages.

1.1.1.1 Message frames

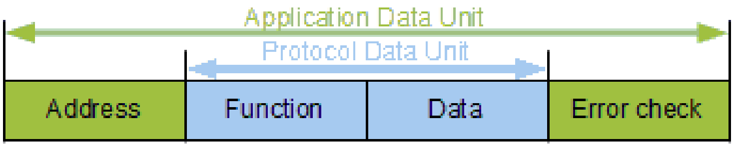
Several Modbus messaging formats ("frames") exist and are used for different purposes and environments, though many of them are not compliant to the Modbus standard. The standard-compliant frame variants are listed in the following table:

Protocol	Description
RTU	Original Modbus standard. Binary data is sent via serial connections such as RS-232 or similar.
ASCII	Similar to RTU. Instead of raw binary, data is encoded in ASCII.
Modbus/TCP	Binary data is encapsulated in a TCP frame and sent via network connections such as Ethernet. This variant can also be used with UDP instead of TCP and is then called Modbus/UDP.

When using ASCII frames or RTU frames via serial connection, parameters such as baud rate and parity bits must be set correctly for all connected devices. When using Modbus/TCP, setting these parameters is not required, but correct IP address and port number are required instead. The standard port number for Modbus/TCP is port 502.

1.1.2 Message fields

Although the different message frames are each handled differently by the protocol, RTU frames and ACSCII frames each include the same four fields. Field 2 and 3 constitute the Protocol Data Unit (PDU), which is part of Modbus/TCP message frames as well, while all 4 fields together constitute the Application Data Unit (ADU):



Field 1 includes the address of a slave device, either indicating the slave that is designated to receive the message from its master, or indicating the slave that sent the message towards its master. This address, which is referred to as “unit ID” or “slave address”, is a number from 1 to 247 and is uniquely assigned to a single slave device, allowing these devices to listen for messages containing their specific ID. Additionally, ID 0 is used to send broadcasts and ID 255 usually is reserved for communications with a Modbus gateway.

Field 2 includes a function code, which, when sent by a master, indicates the instruction a slave is asked to carry out. When sent by a slave, on the other hand, the function code indicates the instruction the slave is responding to.

Field 3 contains variable amounts of data, e.g. certain data addresses a master wants a slave to read, or the data a slave is reporting towards its master.

In field 4 Modbus messages carry a checksum to allow their respective recipients to determine whether a message has arrived completely.

RTU message frames

When using RTU frames, each byte contained in a message is sent as binary data. The main advantage of this mode is its greater density, allowing better data throughput for the same baud rate when compared to ASCII frames. To indicate the start of an RTU frame, the ADU is preceded by a silent interval of at least 3.5 Byte times, hence the length of that interval depends on the configuration of the devices in use. To indicate the end of a frame, another silent interval of 3.5 Byte times succeeds the ADU. Note that one single interval of silence can, at the same time, indicate the end of one frame and the beginning of another frame. RTU frames use Cyclic Redundancy Checks (CRC).

A complete RTU frame can be depicted as shown below:

Start	Address	Function	Data	Error check	End
3.5 Bytes (silence)	1 Byte	1 Byte	n Bytes	2 Bytes (CRC)	3.5 Bytes (silence)

ASCII message frames

When using ASCII frames, each byte contained in a message is encoded and sent as two ASCII characters. This allows time intervals of up to one second to occur between characters without causing an error. To indicate the start of a frame, the ADU is preceded by a single character, which always is a colon (`0x3A`). To indicate the end of a frame, another two trailing characters succeed the ADU, which always are “Carriage Return” and “Line Feed” (`0x0D` and `0x0A`, respectively). ASCII frames use Longitudinal Redundancy Checks (LRC).

A complete ASCII frame can be depicted as shown below:

Start	Address	Function	Data	Error check	End
1 char (:)	2 chars	2 chars	n chars	1 char (LRC)	2 chars (CR, LF)

Modbus/TCP message frames

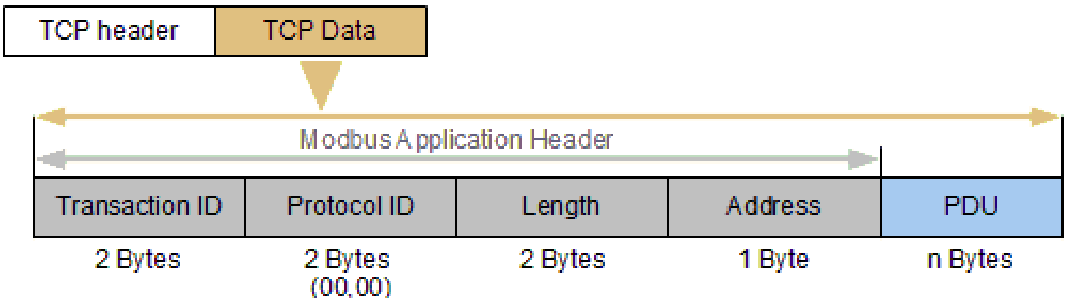
When using Modbus/TCP frames, an additional header called "Modbus Application Header" precedes the PDU. Its four fields contain the transaction ID, the protocol ID, the length of the following frame and the slave address.

The transaction ID is a number from 0 to 65,535 encoded into two bytes. A master device will increment this number for every request it sends to a slave, while slaves simply echoe the number back to their master. By doing so, the master is able to decide wether messages got lost or delayed in transmission.

The protocol ID is a two-byte value, too, but is always 00 00. The length field consists of two more bytes indicating the length of the remaining message.

Finally the address field contains a unit ID, similar to that included in ASCII frames or RTU frames. But with Modbus/TCP, it does not necessarily serve a purpose, as the IP address is used instead to indicate the message's recipient. However, the unit ID is still part of the message and might be used to decide whether a device forwards a message onto a serial connection, thereby allowing devices without networking capabilities to be used in these environments, too.

A complete Modbus/TCP frame can be depicted as shown below:



1.1.3 Modbus data basics

Modbus was specifically designed for usage in supervisory control and data acquisition systems, connecting a supervisory computer with one or several remote terminal units (RTU). Therefore, data types used in Modbus communications have been named according to that implementation. When the Modbus protocol was extended in 1999 to include TCP frames via Ethernet, the data types' names were left unchanged.

Four primary data types are used by Modbus:

Data type	Description
`Coil`	single bit, alterable by an application program, read-write
`Discrete Input`	single bit, provided by an I/O system, read-only
`Holding Register`	16-bit, alterable by an application program, read-write
`Input Register`	16-bit, provided by an I/O system, read-only

For referencing data, Modbus uses a concept of data tables, which are arrays or blocks of memory used to store data. This data can then be referenced by using data table addresses, represented by simple integer values between 0 and 65,535. While it is fully standard-compliant to implement up to 65,536 addresses for each data type, the number of addresses implemented in a particular device usually is much lower. Therefore, Modbus implementations might even assign specific address ranges of a single table to each type of data. While the Modbus standard itself does not specify distinct address ranges, typical Modbus implementations utilize the following assignments:

- 0xxxx-ranged addresses store coils.
- 1xxxx-ranged addresses store discrete inputs.
- 3xxxx-ranged addresses store input registers.
- 4xxxx-ranged addresses store holding registers.

Modbus uses a big-endian representation for data table addresses as well as for the actual data itself. Therefore, the most significant byte is sent first when a numerical quantity larger than a single byte is transmitted. For example

- (16-bits) `0x1234` gets sent as `0x12 0x34`, and
- (32-bits) `0x12345678` gets sent as `0x12 0x34 0x56 0x78`.

In addition to single bit data types (e.g. representing Boolean values) and 16-bit data types (e.g. representing integers), it is also possible to use large data types such as long integers, floating point numbers and strings by splitting them over several addresses. However, the Modbus standard does not stipulate this, hence it is up to the individual user to split and store data accordingly.

1.1.4 emModbus data handling

While Modbus data is always used in big endian emModbus takes addresses and values to set in registers and values read in host endianess to make your life easier. The data will then be converted by emModbus so you do not have to do it on your own.

This is easy for register accesses which are U16 registers. This means that you can simply use an U16 variable with emModbus for reading/writing a register.

For coils emModbus expects and delivers data as stream of bytes with the bits in each byte treated in LSB order. For a register base addr. of 1000 this means:

- Bit 0 of the first byte means value of coil at addr. 1000.
- Bit 1 of the first byte means value of coil at addr. 1001.
- Bit 7 of the first byte means value of coil at addr. 1007.
- Bit 1 of the second byte means value of coil at addr. 1008.
- Bit 7 of the second byte means value of coil at addr. 1015.

The byte order itself is treated in MSB by emModbus.

1.1.5 Further reading

This guide explains the usage of the emModbus stack. It describes all functions which are required to build a Modbus application. For a deeper understanding of the official Modbus protocol, please visit:

- Modbus Organization official website: http://www.modbus.org/

1.2 emModbus

emModbus is written in ANSI C and can be used on virtually any CPU. It combines a maximum of performance with a small memory footprint and comes with all features typically required by embedded systems. RAM usage has been kept to a minimum by smart buffer handling.

1.2.1 Features of emModbus

Features of emModbus include:

- Easy to integrate.
- Low memory footprint.
- ANSI-C code is completely portable and runs on any target.
- Follows the SEGGER coding standards: Efficient and compact, yet easy to read, understand & debug.
- Supports ASCII, RTU and Modbus/TCP (and UDP) protocol.
- Sample applications for all protocols included.
- Kernel abstraction layer: can be used with or without any RTOS.
- Works out-of-the-box with embOS.
- Modbus/TCP can be used with standard socket interface and any TCP/IP stack.
- Works out-of-the-box with embOS/IP.
- Project for executable on PC for Microsoft Visual Studio available.

The following table shows the contents of the emModbus root directory:

Directory	Content
`Application\*.c`	Contains example applications to run emModbus with UART or embOS/IP.
`Config\*.c`	Contains the emModbus configuration files. Refer to *Configuring emModbus* on page 87 for further information.
`MB\*.c`	Contains the emModbus sources such as `MB_Core.c`, `MB_CHANNEL.c` , `MB_MASTER.c` and `MB_SLAVE.c`
`SEGGER\*.c`	Contains optimized memcpy routines to speed up the stack.
`Windows\*.c`	Contains the source(s), project file(s) and a executable(s) to run emModbus on a Microsoft Windows host.

1.2.2 emModbus requirements

TCP/IP stack

For usage of Modbus/TCP, emModbus requires a TCP/IP capable stack. emModbus can be used with any TCP/IP stack that supports BSD Standard Sockets. The shipment includes an implementation which uses the socket API of embOS/IP.

Multi tasking

Although emModbus can be used completely without an RTOS, it is recommended to use emModbus in a multi tasking system, at least when implementing a Modbus master.

1.2.3 Development environment (compiler)

The CPU used is of no importance; only an ANSI-compliant compiler complying with at least one of the following international standard is required:

- ISO/IEC/ANSI 9899:1990 (C90) with support for C++ style comments (//)
- ISO/IEC 9899:1999 (C99)
- ISO/IEC 14882:1998 (C++)

If your compiler has some limitations, let us know and we will inform you if these will be a problem when compiling the software. Any compiler for 16/32/64-bit CPUs or DSPs that we know of can be used; most 8-bit compilers can be used as well.

A C++ compiler is not required, but can be used. The application program can therefore also be programmed in C++ if desired.

1.3 Tasks and interrupt usage

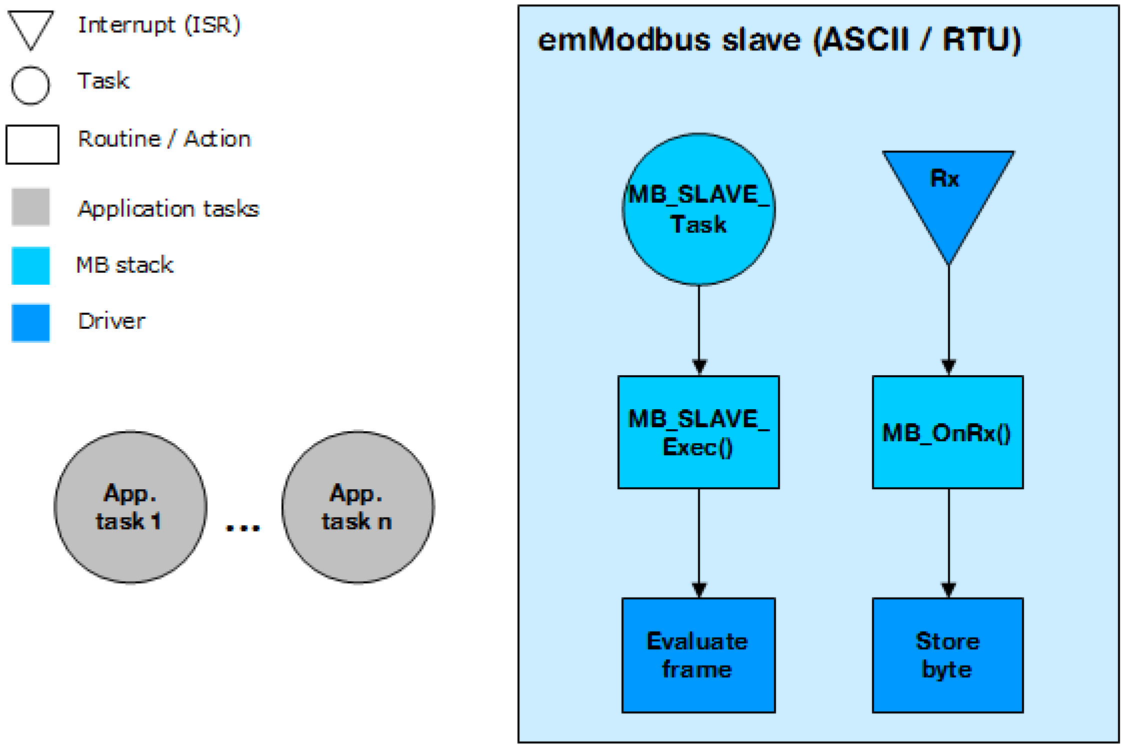
emModbus can be used in an application in two different ways.

- With tasks dedicated to the stack.
- Without tasks dedicated to the stack.

The following chapters provide information on these ways for both ASCII and RTU frames as well as for Modbus/TCP (or UDP) frames.

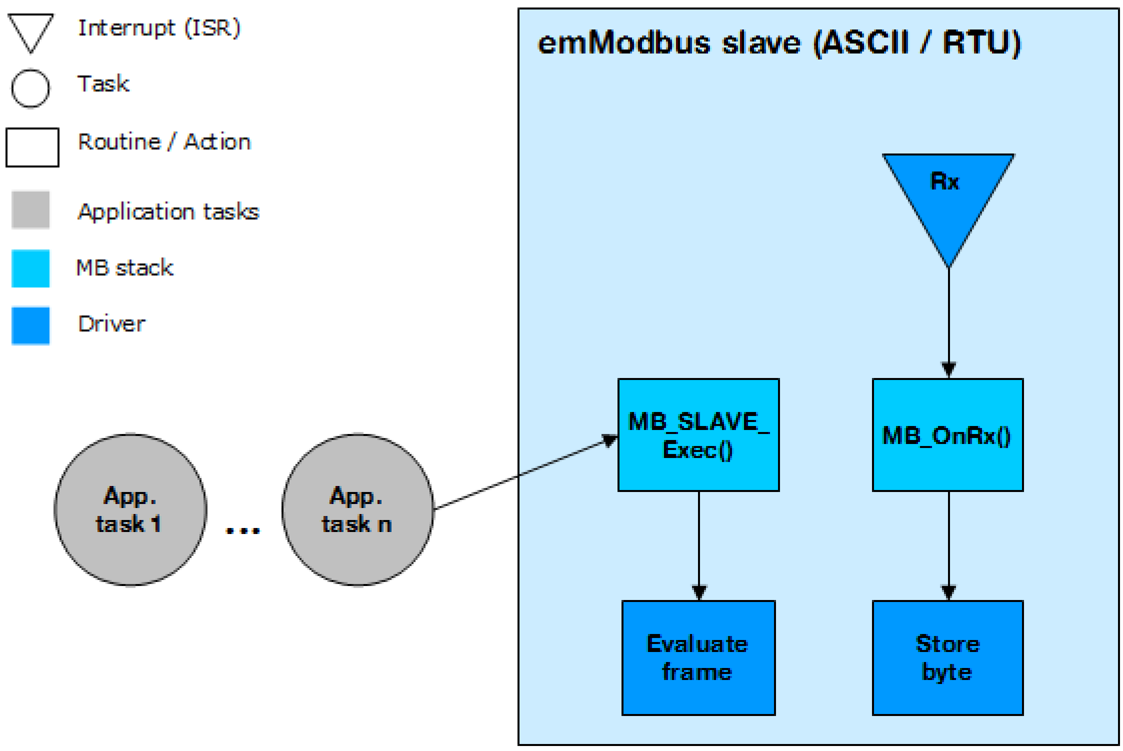
1.3.1 ASCII / RTU slave with tasks dedicated to the stack

To use tasks dedicated to the stack is the simplest way to use emModbus with ASCII and/or RTU frames. The `MB_SLAVE_Task` handles housekeeping operations and evaluation of incoming frames. The "Store byte" operation is called and performed from within the Interrupt Service Routine, hence no additional task is required.



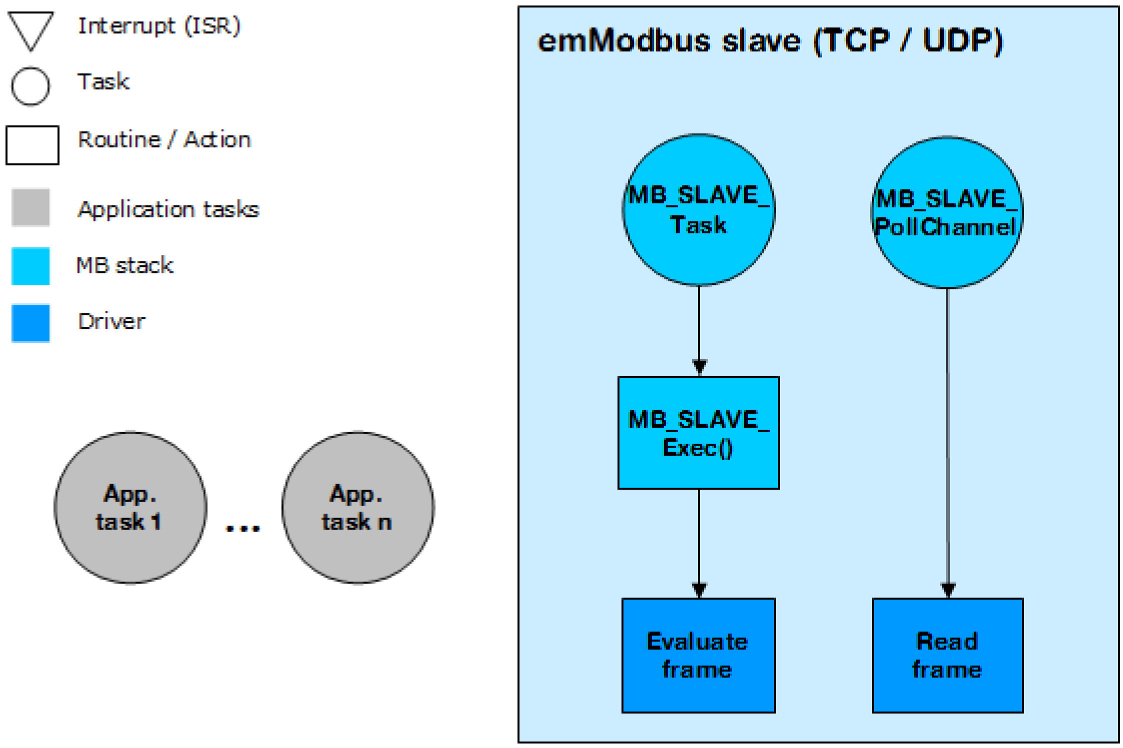
1.3.2 ASCII / RTU slave without tasks dedicated to the stack

emModbus ASCII and/or RTU frames can also be used without any task dedicated to the stack, if an application task calls `MB_SLAVE_Exec()` periodically. The "Store byte" operation is called and performed from within the Interrupt Service Routine.



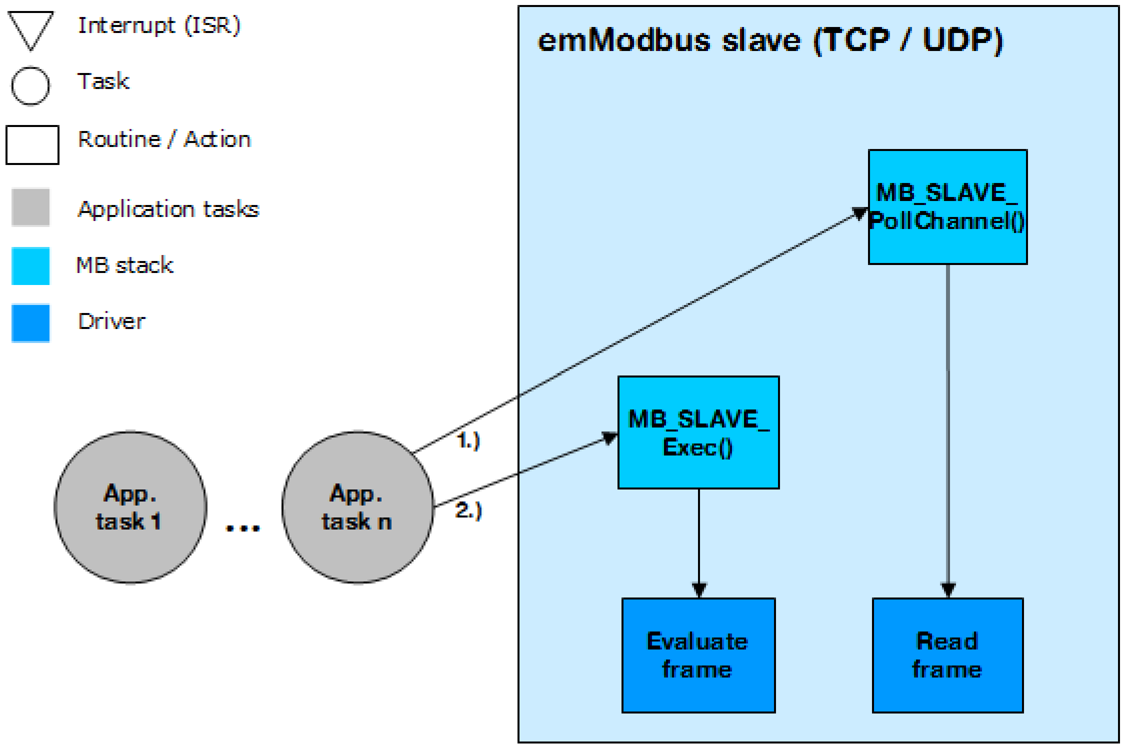
1.3.3 TCP / UDP slave with tasks dedicated to the stack

To use tasks dedicated to the stack is the simplest way to use emModbus/TCP. The `MB_SLAVE_Task` handles housekeeping operations and evaluation of incoming frames. The "Read frame" operation is called and performed by another task, `MB_SLAVE_PollChannel`, which periodically polls for incoming frames.



1.3.4 TCP / UDP slave without tasks dedicated to the stack

emModbus/TCP can also be used without any task dedicated to the stack, if an application task consecutively calls `MB_SLAVE_PollChannel()` and `MB_SLAVE_Exec()` periodically.



1.3.5 emModbus master

The emModbus master API is independent of the usage of any real-time operating system. However, by utilizing an RTOS the emModbus interface becomes easier to use and more comfortable to integrate into any desired application.

Chapter 2

Getting Started

The first step in getting started with emModbus is to compile it for and run it on the target system. This chapter explains how to do this.

In this document the IAR Embedded Workbench® IDE is used for all examples and screenshots, but every other ANSI-C toolchain can be used as well. It is also possible to use makefiles; in this case, "add to the project" translates into "add to the makefile".

2.1 Installation

emModbus is typically shipped as a .ZIP file in electronic form. In order to install emModbus, extract it to any folder of your choice, preserving the directory structure of the .ZIP file.

To create a running emModbus project, there are three different routes available:

- Upgrade a trial version by adding source code.
- Upgrade an embOS start project.
- Create a project from scratch.

The following example procedures describe each of these routes. They focus on integrating an emModbus slave device using Modbus/TCP frames, but any other emModbus project can be created as well by following the same steps.

emModbus via TCP is optimized for use with embOS/IP, SEGGER's TCP/IP stack. However, emModbus can be used with any other TCP/IP stack as well. Note that when using ASCII frames or RTU frames, the integration of a TCP/IP stack is not required and should be omitted for smaller code size. Similarly, if no real-time operating system is required, the integration of an RTOS should be omitted as well.

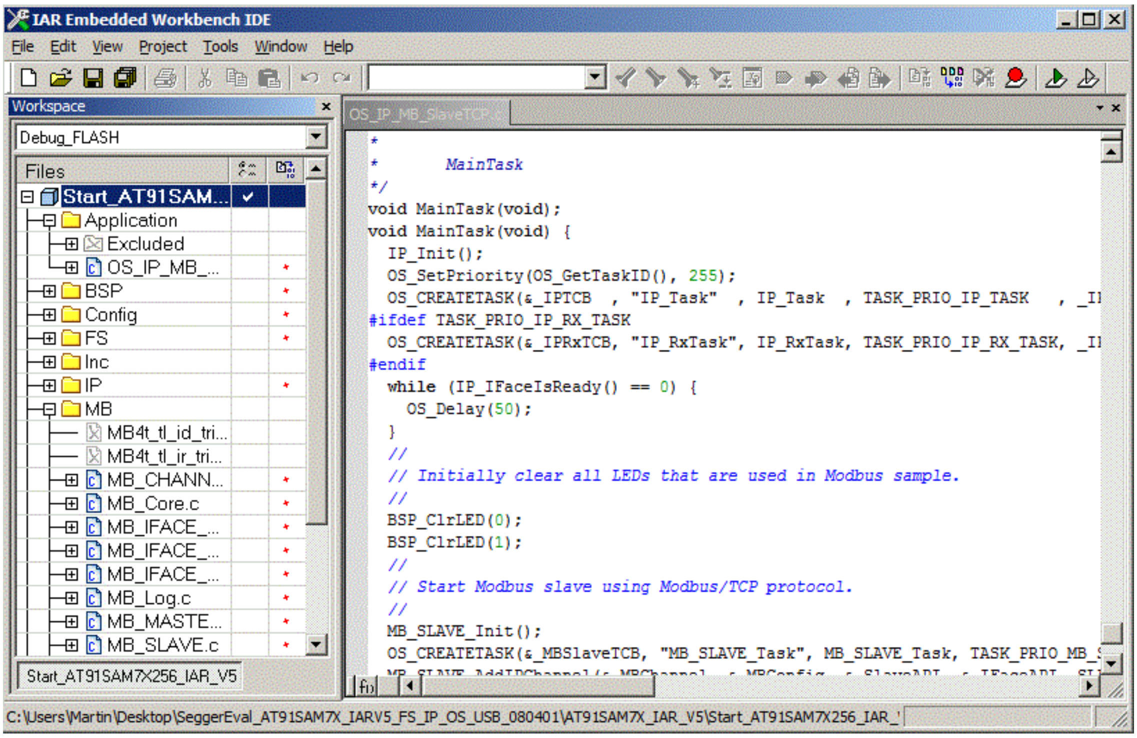
2.2 Upgrade a trial version

Various trial packages for different target hardware are available at SEGGER's website.

Note that not all trial packages currently available contain a trial of emModbus. If you are interested in a specific package that does not contain emModbus yet, feel free to contact us. Including emModbus in a trial package can be completed quickly by our Expert Team. Additionally, trial packages that do not contain embOS/IP do lack an appropriate TCP/IP stack, which is required for Modbus/TCP frames. However, ASCII frames and RTU frames might be used regardless of a TCP/IP stack.

Replace libraries with sources

After downloading the trial package, extract the project contained in the .ZIP file to any folder of your choice and open the workspace/project file. Copy the source files from the folder MB of your emModbus shipment into the folder MB of your downloaded package, add the files to the project and exclude the trial libraries from the build.



Build the project

Build the project; it should compile without errors and warnings. If any problem is encountered during the build process, checking the include paths and project configurations is advisable as first step. When building completes, download the output into the designated target and start the application.

Test the project

We recommend testing emModbus devices by using their respective counterparts, e.g. using a emModbus/TCP master to test an emModbus/TCP slave and vice versa. Alternatively, devices can also be tested with a desktop computer running an appropriate Modbus application.

Refer to *Testing emModbus applications* on page 98 for additional information.

2.3 Upgrade an embOS start project

Begin with a sample project for embOS, SEGGER's real-time operating system, then include embOS/IP and emModbus into the project.

The emModbus default configuration is preconfigured with valid values, which match the requirements of most applications.

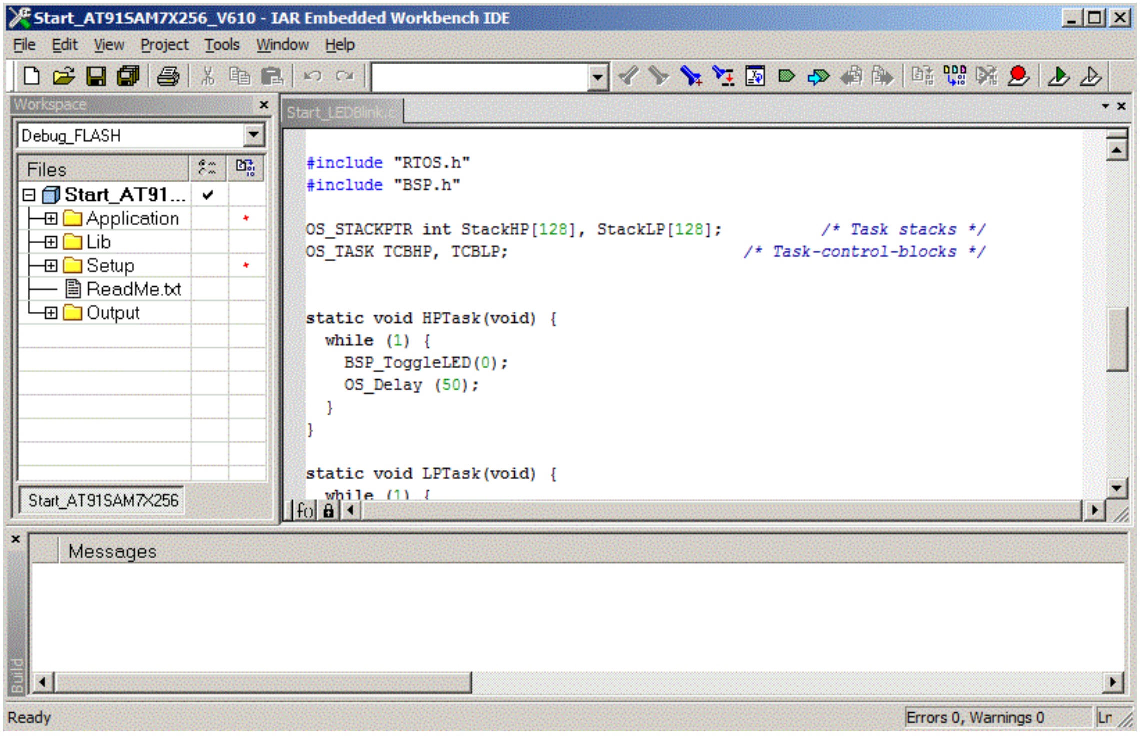
Procedure to follow

Integration of emModbus is a relatively simple process, which consists of the following steps:

1. Open an embOS start project.
2. Add embOS/IP to the start project.
3. Add emModbus to the start project.
4. Build the project.

2.3.1 Step 1: Open an embOS start project

We recommend that you use one of the supplied embOS start projects for your target system. Compile the project and run it on your target hardware.



2.3.2 Step 2: Adding embOS/IP to the start project

Add all source files in the following directories to your project:

- `Config`
- `IP`
- `SEGGER` (optional)

The `Config` folder includes all configuration files of embOS/IP. The configuration files are preconfigured with valid values that match the requirements of most applications. Add the hardware configuration `IP_Config_<TargetName>.c` supplied with the driver shipment.

If your hardware is currently not supported, use the example configuration file and the driver template to write your own driver. The example configuration file and the driver template is located in the `Sample` folder.

The `SEGGER` folder is an optional component of the embOS/IP shipment. It contains optimized MCU and/or compiler specific files, for example an optimized memcopy function.

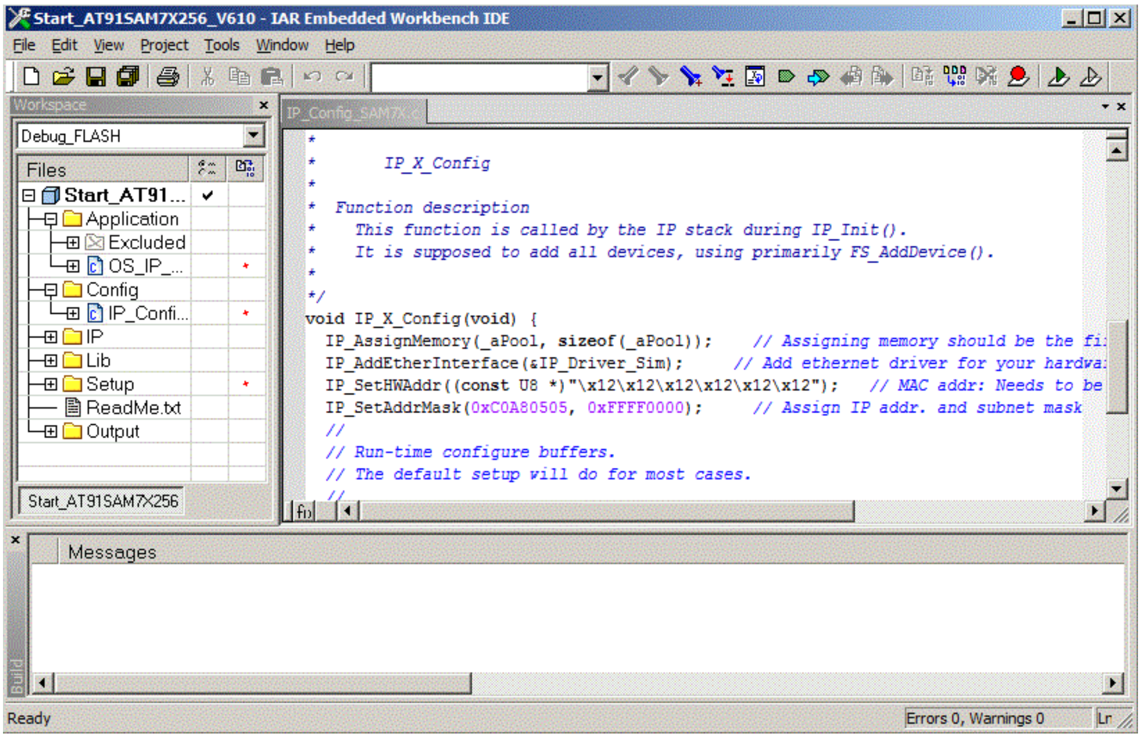
Replace BSP.c and BSP.h of your embOS start project

Replace the BSP.c source file and the BSP.h header file used in your embOS start project with the one which is supplied with the embOS/IP shipment. Some drivers require a special functions which initializes the network interface of the driver. This function is called `BSP_ETH_Init()`. It is used to enable the ports which are connected to the network hardware. All network interface driver packages include the `BSP.c` and `BSP.h` files irrespective if the `BSP_ETH_Init()` function is implemented.

Configuring the include path

The include path is the path in which the compiler looks for include files. In cases where the included files (typically header files, `.h`) do not reside in the same directory as the C file to compile, an include path needs to be set. In order to build the project with all added files, you will need to add the following directories to your include path:

- `Config`
- `Inc`
- `IP`



2.3.3 Step 3: Adding emModbus to the start project

Add all source files in the following directories to your project:

- `Config`
- `MB`
- `SEGGER` (optional)

The `Config` folder includes all configuration files of emModbus. The configuration files are preconfigured with valid values, which match the requirements of most applications.

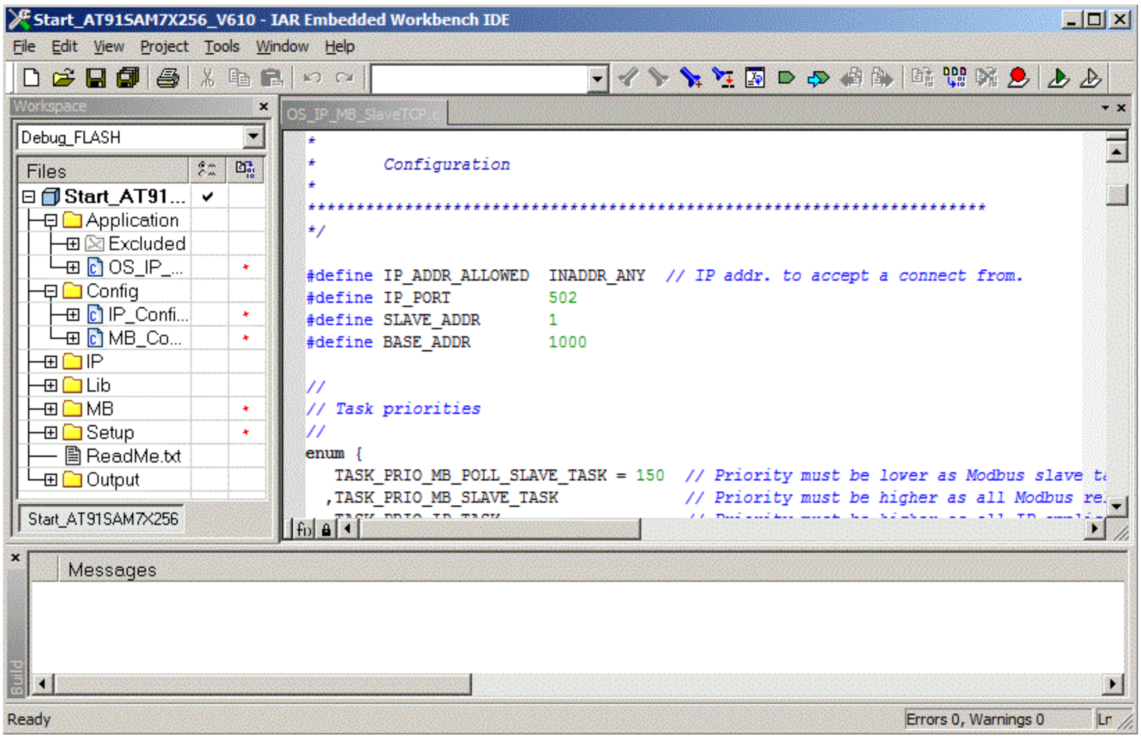
Configuring the include path

The include path is the path in which the compiler looks for include files. In cases where the included files (typically header files, `.h`) do not reside in the same directory as the C file to compile, an include path needs to be set. In order to build the project with all added files, you will need to add the following directories to your include path:

- `Config`
- `MB`

Select the start application

For quick and easy testing of your emModbus integration, start with the code found in the folder `Application`. Add one of the applications to your project (for example `OS_IP_MB_SlaveTCP.c`).



2.3.4 Step 4: Build the project

Build the project

Build the project; it should compile without errors and warnings. If any problem is encountered during the build process, checking the include paths and project configurations is advisable as first step. When done building, download the output into the designated target and start the application.

Test the project

We recommend testing emModbus devices by using their respective counterparts, e.g. using a emModbus/TCP master to test an emModbus/TCP slave and vice versa. Alternatively, devices can also be tested with a desktop computer running an appropriate Modbus application.

Refer to *Testing emModbus applications* on page 98 for additional information.

2.4 Create a project from scratch

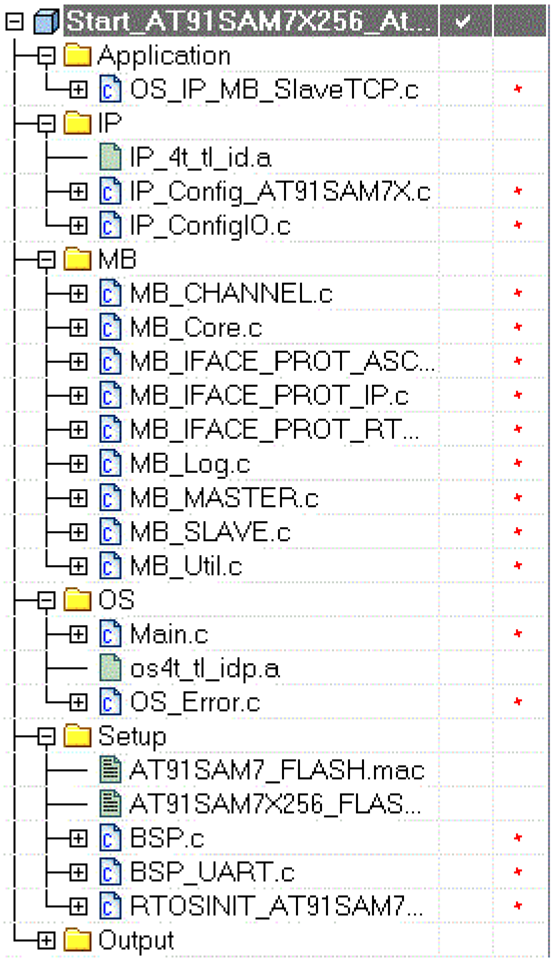
To create a project from scratch, some steps have to be taken:

- A project or make file has to be created for the specific toolchain.
- The project configurations may need adjustments.
- The hardware routines have to be implemented.
- The path of any required header files has to be set as include path.

To get the target up and running is a lot easier if target hardware drivers are already available. In that case, these drivers can be used.

Creating the project or make file

The screenshot below gives an idea about a possible project setup:



Build the project

Build the project; it should compile without errors and warnings. If any problem is encountered during the build process, checking the include paths and project configurations is advisable as first step. When done building, download the output into the designated target and start the application.

Test the project

We recommend testing emModbus devices by using their respective counterparts, e.g. using a emModbus/TCP master to test an emModbus/TCP slave and vice versa. Alternatively, devices can also be tested with a desktop computer running an appropriate Modbus application.

Refer to *Testing emModbus applications* on page 98 for additional information.

Chapter 3

Example applications

In this chapter, you will find a description of the emModbus example applications that are delivered together with the emModbus shipment.

3.1 Overview

Example applications for emModbus are supplied in source code in the Application folder. These can be used for testing the correct installation and proper function of the device running emModbus.

The following start application files are provided:

File	Description
`OS_IP_MB_MasterTCP.c`	Demonstrates the functionality of a Modbus Master device using TCP frames via network.
`OS_IP_MB_SlaveTCP.c`	Demonstrates the functionality of a Modbus Slave device using TCP frames via network.
`OS_MB_MasterASCII.c`	Demonstrates the functionality of a Modbus Master device using ASCII frames via serial connection.
`OS_MB_MasterRTU.c`	Demonstrates the functionality of a Modbus Master device using RTU frames via serial connection.
`OS_MB_SlaveASCII.c`	Demonstrates the functionality of a Modbus Slave device using ASCII frames via serial connection.
`OS_MB_SlaveRTU.c`	Demonstrates the functionality of a Modbus Slave device using RTU frames via serial connection.

3.1.1 OS_IP_MB_MasterTCP.c

This sample demonstrates emModbus master functionalities using the Modbus/TCP protocol. It opens a channel and tries to establish a TCP connection to a Modbus slave device, which is known to the master by the slave's IP address as defined at the beginning of the file. The master also uses a given port for this connection, which is defined at the beginning of the file (e.g. port 502, the standard port for Modbus communications). When a connection is established, the master repeatedly sends queries to the slave, asking it to perform the function "write coil", and waits for appropriate responses.

3.1.2 OS_IP_MB_SlaveTCP.c

This sample demonstrates emModbus slave functionalities using the Modbus/TCP protocol. It opens a channel and waits for incoming TCP connections on a given port, which is known to the slave as defined at the beginning of the file. When an incoming connection from a Modbus master device has been established, the slave reacts to queries it receives from the master. When ordered to write a coil (like the associated Modbus master sample does), the slave will toggle LEDs to signal its new status.

3.1.3 OS_MB_MasterASCII.c

This emModbus sample demonstrates emModbus master functionalities using ASCII frames. It opens a channel and repeatedly sends queries to a Modbus slave device (specified by its slave ID as defined at the beginning of the file), asking it to perform the function "write coil", and waits for appropriate responses.

3.1.4 OS_MB_MasterRTU.c

This emModbus sample demonstrates emModbus master functionalities using RTU frames. It opens a channel and repeatedly sends queries to a Modbus slave device (specified by its slave ID as defined at the beginning of the file), asking it to perform the function "write coil", and waits for appropriate responses.

3.1.5 OS_MB_SlaveASCII.c

This sample demonstrates emModbus slave functionalities using ASCII frames. It opens a channel and waits for incoming queries from a Modbus master device. When ordered to write a coil (like the associated Modbus master sample does), the slave will toggle LEDs to signal its new status.

3.1.6 OS_MB_SlaveRTU.c

This sample demonstrates emModbus slave functionalities using RTU frames. It opens a channel and waits for incoming queries from a Modbus master device. When ordered to write a coil (like the associated Modbus master sample does), the slave will toggle LEDs to signal its new status.

Chapter 4

emModbus API

In this chapter, you will find a description of each emModbus core function.

4.1 API functions

The table below lists the available API functions within their respective categories.

Function	Description
Channel specific core functions	
MB_CHANNEL_Disconnect()	Disconnect a previously connected channel, if the channel did any connect at all and is currently connected.
Master specific core functions	
MB_MASTER_AddASCIIChannel()	Adds a master channel that uses the Modbus ASCII protocol via a serial connection.
MB_MASTER_AddIPChannel()	Adds a master channel that uses the Modbus/TCP or Modbus/UDP protocol via IP.
MB_MASTER_AddRTUChannel()	Adds a master channel that uses the Modbus RTU protocol via a serial connection.
MB_MASTER_DeInit()	De-Initializes the master channels, resources and removes the master endpoint from the global endpoint list.
MB_MASTER_Init()	Initializes the master resources and adds the master endpoint to the global endpoint list.
Master instruction set	
MB_MASTER_ReadCoils()	Reads coils from a slave.
MB_MASTER_ReadDI()	Reads Discrete Inputs from a slave.
MB_MASTER_ReadHR()	Reads Holding Registers from a slave.
MB_MASTER_ReadIR()	Reads Input Registers from a slave.
MB_MASTER_WriteCoil()	Writes a single coil to a slave.
MB_MASTER_WriteCoils()	Writes multiple coils to a slave.
MB_MASTER_WriteReg()	Writes a single register to a slave.
MB_MASTER_WriteRegs()	Writes multiple registers to a slave.
Slave specific core functions	
MB_SLAVE_AddASCIIChannel()	Adds a slave channel that uses the Modbus ASCII protocol via a serial connection.
MB_SLAVE_AddIPChannel()	Adds a slave channel that uses the Modbus/TCP or Modbus/UDP protocol via IP.
MB_SLAVE_AddRTUChannel()	Adds a slave channel that uses the Modbus RTU protocol via a serial connection.
MB_SLAVE_ConfigIgnoreSlaveAddr()	Function that configures if the slave address is ignored.
MB_SLAVE_DeInit()	De-Initializes the slave channels, resources and removes the slave endpoint from the global endpoint list.
MB_SLAVE_Exec()	Loops once over all slave channels and processes data when the channel has been signalled ready.
MB_SLAVE_Init()	Initializes the slave resources and adds the slave endpoint to the global endpoint list.
MB_SLAVE_PollChannel()	Function that has to be periodically polled for each slave channel that requires re-

Function	Description
	questing data instead of getting it via interrupt.
MB_SLAVE_SetCustomFunctionCodeHandler()	Sets a handler for custom function codes.
MB_SLAVE_Task()	Wrapper function that runs MB_SLAVE_Exec() in a task.
Other core functions	
MB_ConfigTimerFreq()	This function allows setting a user defined timer frequency instead of the default 1kHz frequency.
MB_OnRx()	Function called by byte oriented transmission channels that receive an interrupt for new data received.
MB_OnTx()	Function called by byte oriented transmission channels once a Tx complete interrupt has been received to send the next byte or report back that there is no more to send.
MB_TimerTick()	Function called on each timer interrupt to manage internal RTU timeout with serial channels using the RTU protocol.
Helper functions	
MB_LoadU16BE()	Loads an U16 value from a big endian memory location.
MB_StoreU16BE()	Stores an U16 value to a big endian memory location.

4.1.1 Channel specific core functions

4.1.1.1 MB_CHANNEL_Disconnect()

Description

Disconnect a previously connected channel, if the channel did any connect at all and is currently connected.

Prototype

```
void MB_CHANNEL_Disconnect(MB_CHANNEL * pChannel);
```

Parameters

Parameter	Description
pChannel	Pointer to element of MB_CHANNEL.

4.1.2 Master specific core functions

4.1.2.1 MB_MASTER_AddASCIIChannel()

Description

Adds a master channel that uses the Modbus ASCII protocol via a serial connection.

Prototype

```
void MB_MASTER_AddASCIIChannel(       MB_CHANNEL            * pChannel,
                                      MB_IFACE_CONFIG_UART  * pConfig,
                                const MB_IFACE_UART_API     * pIFaceAPI,
                                      U32                     Timeout,
                                      U8                      SlaveAddr,
                                      U32                     Baudrate,
                                      U8                      DataBits,
                                      U8                      Parity,
                                      U8                      StopBits,
                                      U8                      Port);
```

Parameters

Parameter	Description
pChannel	Pointer to element of MB_CHANNEL that is added to linked list.
pConfig	Pointer to element of MB_IFACE_CONFIG_UART.
pIFaceAPI	Pointer to element of MB_IFACE_UART_API used to read/write from/to interface.
Timeout	Timeout [ms] to wait for answer.
SlaveAddr	Slave addr. to listen on.
Baudrate	Desired baudrate to configure.
DataBits	Number of data bits used in UART protocol.
Parity	Parity used in UART protocol.
StopBits	Number of stop bits used in UART protocol.
Port	UART port number used by this channel.

Example

```
//
// Static declarations
//
static       MB_CHANNEL           _MBChannel;
static       MB_IFACE_CONFIG_UART _MBConfig;
static const MB_IFACE_UART_API    _IFaceAPI = {
  _SendByte, _Init, _DeInit, NULL, NULL, NULL, NULL, NULL, NULL
};

//
// Code running in its own task
//
static void _MasterTask(void) {
  MB_MASTER_Init();                    // Init master
  MB_MASTER_AddASCIIChannel(&_MBChannel, &_MBConfig, &_IFaceAPI, 3000, 1,
               38400, 8, 0, 1, 0);     // Add master channel
  do { ... }                           // e.g. master/slave communications
}
```


4.1.2.2 MB_MASTER_AddIPChannel()

Description

Adds a master channel that uses the Modbus/TCP or Modbus/UDP protocol via IP.

Prototype

```
void MB_MASTER_AddIPChannel(      MB_CHANNEL          * pChannel,
                                  MB_IFACE_CONFIG_IP  * pConfig,
                            const MB_IFACE_IP_API     * pIFaceAPI,
                                  U32                   Timeout,
                                  U8                    SlaveAddr,
                                  U32                   IPAddr,
                                  U16                   Port);
```

Parameters

Parameter	Description
pChannel	Pointer to element of MB_CHANNEL that is added to linked list.
pConfig	Pointer to element of MB_IFACE_CONFIG_IP.
pIFaceAPI	Pointer to element of MB_IFACE_IP_API used to read/write from/to interface.
Timeout	Timeout [ms] to wait for answer.
SlaveAddr	Slave addr. to access.
IPAddr	IP addr. of slave.
Port	Slave port to connect to.

Example

```
//
// Static declarations
//
static       MB_CHANNEL         _MBChannel;
static       MB_IFACE_CONFIG_IP _MBConfig;
static const MB_IFACE_IP_API    _IFaceAPI = {
  NULL, NULL, NULL, _Send, _Recv, _Connect, _Disconnect, NULL, NULL
};

//
// Code running in its own task
//
static void _MasterTask(void) {
  MB_MASTER_Init();                             // Init master
  MB_MASTER_AddIPChannel(&_MBChannel, &_MBConfig, &_IFaceAPI, 3000, 1,
           IP_BYTES2ADDR(192,168,1,80), 502);   // Add master channel
  do { ... }                                    // e.g. master/slave communications
}
```


4.1.2.3 MB_MASTER_AddRTUChannel()

Description

Adds a master channel that uses the Modbus RTU protocol via a serial connection.

Prototype

```
void MB_MASTER_AddRTUChannel(       MB_CHANNEL             * pChannel,
                                    MB_IFACE_CONFIG_UART * pConfig,
                              const MB_IFACE_UART_API     * pIFaceAPI,
                                    U32                      Timeout,
                                    U8                       SlaveAddr,
                                    U32                      Baudrate,
                                    U8                       DataBits,
                                    U8                       Parity,
                                    U8                       StopBits,
                                    U8                       Port);
```

Parameters

Parameter	Description
pChannel	Pointer to element of MB_CHANNEL that is added to linked list.
pConfig	Pointer to element of MB_IFACE_CONFIG_UART.
pIFaceAPI	Pointer to element of MB_IFACE_UART_API used to read/write from/to interface.
Timeout	Timeout [ms] to wait for answer.
SlaveAddr	Slave addr. to listen on.
Baudrate	Desired baudrate to configure.
DataBits	Number of data bits used in UART protocol.
Parity	Parity used in UART protocol.
StopBits	Number of stop bits used in UART protocol.
Port	UART port number used by this channel.

Example

```
//
// Static declarations
//
static       MB_CHANNEL           _MBChannel;
static       MB_IFACE_CONFIG_UART _MBConfig;
static const MB_IFACE_UART_API    _IFaceAPI = {
  _SendByte, _Init, _DeInit, NULL, NULL, NULL, NULL, _InitTimer, _DeInitTimer
};

//
// Code running in its own task
//
static void _MasterTask(void) {
  MB_MASTER_Init();                  // Init master
  MB_MASTER_AddRTUChannel(&_MBChannel, &_MBConfig, &_IFaceAPI, 3000, 1,
              38400, 8, 0, 1, 0);    // Add master channel
  do { ... }                         // e.g. master/slave communications
}
```


4.1.2.4 MB_MASTER_DeInit()

Description

De-Initializes the master channels, resources and removes the master endpoint from the global endpoint list.

Prototype

```
void MB_MASTER_DeInit(void);
```


4.1.2.5 MB_MASTER_Init()

Description

Initializes the master resources and adds the master endpoint to the global endpoint list.

Prototype

```
void MB_MASTER_Init(void);
```


4.1.3 Master instruction set

4.1.3.1 MB_MASTER_ReadCoils()

Description

Reads coils from a slave.

Prototype

```
int MB_MASTER_ReadCoils(MB_CHANNEL * pChannel,
                        U8         * pData,
                        U16          Addr,
                        U16          NumItems);
```

Parameters

Parameter	Description
pChannel	Pointer to channel configured to interface a slave.
pData	Pointer to application buffer where to store the read data.
Addr	Address in slave where to find the coils to access.
NumItems	Number of items to read.

Return value

< 0 Error.
= 0 O.K.

Example

```
//
// Static declarations
//
static int        _Result;
static MB_CHANNEL _MBChannel;
static U8         _Data;

//
// Code running in its own task
//
static void _MasterTask(void) {
  MB_MASTER_Init();                                        // Init master
  MB_MASTER_AddASCIIChannel( ... );                        // Add master channel
  _Data   = 0;
  _Result = MB_MASTER_ReadCoils(&_MBChannel, &_Data, 1000, 2); // Read Coils
}
```

4.1.3.2 MB_MASTER_ReadDI()

Description

Reads Discrete Inputs from a slave.

Prototype

```
int MB_MASTER_ReadDI(MB_CHANNEL * pChannel,
                     U8         * pData,
                     U16          Addr,
                     U16          NumItems);
```

Parameters

Parameter	Description
pChannel	Pointer to channel configured to interface a slave.
pData	Pointer to application buffer where to store the read data.
Addr	Address in slave where to find the inputs to access.
NumItems	Number of items to read.

Return value

< 0 Error.
= 0 O.K.

Example

```
//
// Static declarations
//
static int        _Result;
static MB_CHANNEL _MBChannel;
static U16        _aData[2];

//
// Code running in its own task
//
static void _MasterTask(void) {
  MB_MASTER_Init();                                         // Init master
  MB_MASTER_AddASCIIChannel( ... );                         // Add master channel
  _Result = MB_MASTER_ReadDI(&_MBChannel,&_aData[0],1000,2);  // Read Discrete Input
}
```


4.1.3.3 MB_MASTER_ReadHR()

Description

Reads Holding Registers from a slave.

Prototype

```
int MB_MASTER_ReadHR(MB_CHANNEL * pChannel,
                     U8         * pData,
                     U16          Addr,
                     U16          NumItems);
```

Parameters

Parameter	Description
pChannel	Pointer to channel configured to interface a slave.
pData	Pointer to application buffer where to store the read data.
Addr	Address in slave where to find the registers to access.
NumItems	Number of items to read.

Return value

< 0 Error.
= 0 O.K.

Example

```
//
// Static declarations
//
static int        _Result;
static MB_CHANNEL _MBChannel;
static U16        _aData[2];

//
// Code running in its own task
//
static void _MasterTask(void) {
 MB_MASTER_Init();                                         // Init master
 MB_MASTER_AddASCIIChannel( ... );                         // Add master channel
 _Result = MB_MASTER_ReadHR(&_MBChannel,&_aData[0],1000,2);  // Read Holding Register
}
```


4.1.3.4 MB_MASTER_ReadIR()

Description

Reads Input Registers from a slave.

Prototype

```
int MB_MASTER_ReadIR(MB_CHANNEL * pChannel,
                     U8         * pData,
                     U16          Addr,
                     U16          NumItems);
```

Parameters

Parameter	Description
pChannel	Pointer to channel configured to interface a slave.
pData	Pointer to application buffer where to store the read data.
Addr	Address in slave where to find the registers to access.
NumItems	Number of items to read.

Return value

< 0 Error.
= 0 O.K.

Example

```
//
// Static declarations
//
static int        _Result;
static MB_CHANNEL _MBChannel;
static U16        _aData[2];

//
// Code running in its own task
//
static void _MasterTask(void) {
  MB_MASTER_Init();                                      // Init master
  MB_MASTER_AddASCIIChannel( ... );                      // Add master channel
  _Result = MB_MASTER_ReadIR(&_MBChannel,&_aData[0],1000,2);  // Read Input Register
}
```


4.1.3.5 MB_MASTER_WriteCoil()

Description

Writes a single coil to a slave.

Prototype

```
int MB_MASTER_WriteCoil(MB_CHANNEL * pChannel,
                        U16          Addr,
                        U8           OnOff);
```

Parameters

Parameter	Description
pChannel	Pointer to channel configured to interface a slave.
Addr	Address in slave where to find the coil to access.
OnOff	Write coil to 0: Off or 1: On .

Return value

< 0 Error.
= 0 O.K.

Example

```
//
// Static declarations
//
static int        _Result;
static MB_CHANNEL _MBChannel;

//
// Code running in its own task
//
static void _MasterTask(void) {
  MB_MASTER_Init();                                  // Init master
  MB_MASTER_AddASCIIChannel( ... );                  // Add master channel
  _Result = MB_MASTER_WriteCoil(&_MBChannel, 1000, 1);  // Write Coil
}
```

4.1.3.6 MB_MASTER_WriteCoils()

Description

Writes multiple coils to a slave.

Prototype

```
int MB_MASTER_WriteCoils(MB_CHANNEL * pChannel,
                         U8         * pData,
                         U16          Addr,
                         U16          NumItems);
```

Parameters

Parameter	Description
pChannel	Pointer to channel configured to interface a slave.
pData	Pointer to application buffer where the values to write are stored.
Addr	Address in slave where to find the coils to access.
NumItems	Number of items to write. Not to be mistaken with number of bytes that will be used for x coils.

Return value

< 0 Error.
= 0 O.K.

Example

```
//
// Static declarations
//
static int        _Result;
static MB_CHANNEL _MBChannel;
static U8         _Data[2];

//
// Code running in its own task
//
static void _MasterTask(void) {
  MB_MASTER_Init();                                              // Init master
  MB_MASTER_AddASCIIChannel( ... );                              // Add master channel
  //
  // Set coils at addr. 1000 & 1003.
  // Clr. coild 1001 - 1002 & 1004 - 1007.
  //
  _Data[0] = (1 << 0) | (1 << 3);
  //
  // Set coil at addr. 1008.
  //
  _Data[1] = (1 << 0);                                           // Add master channel
  _Result = MB_MASTER_WriteCoils(&_MBChannel, &_Data[0], 1000, 9);  // Write Coils
}
```


4.1.3.7 MB_MASTER_WriteReg()

Description

Writes a single register to a slave.

Prototype

```
int MB_MASTER_WriteReg(MB_CHANNEL * pChannel,
                       U16          Data,
                       U16          Addr);
```

Parameters

Parameter	Description
pChannel	Pointer to channel configured to interface a slave.
Addr	Address in slave where to find the register to access.
Data	16-bit data to write to register.

Return value

< 0 Error.
= 0 O.K.

Example

```
//
// Static declarations
//
static int        _Result;
static MB_CHANNEL _MBChannel;
static U16        _Data;

//
// Code running in its own task
//
static void _MasterTask(void) {
  MB_MASTER_Init();                                        // Init master
  MB_MASTER_AddASCIIChannel( ... );                        // Add master channel
  _Result = MB_MASTER_WriteReg(&_MBChannel, _Data, 1000);  // Write Register
}
```

4.1.3.8 MB_MASTER_WriteRegs()

Description

Writes multiple registers to a slave.

Prototype

```
int MB_MASTER_WriteRegs(MB_CHANNEL * pChannel,
                        U16        * pData,
                        U16          Addr,
                        U16          NumItems);
```

Parameters

Parameter	Description
pChannel	Pointer to channel configured to interface a slave.
pData	Pointer to application buffer where the values to write are stored.
Addr	Address in slave where to find the coils to access.
NumItems	Number of items to write.

Return value

< 0 Error.
= 0 O.K.

Example

```
//
// Static declarations
//
static int        _Result;
static MB_CHANNEL _MBChannel;
static U16        _aData[2];

//
// Code running in its own task
//
static void _MasterTask(void) {
  MB_MASTER_Init();                                       // Init master
  MB_MASTER_AddASCIIChannel( ... );                       // Add master channel
   _Data[0] = 0x0001;
   _Data[1] = 0x1234;
   _Result  = MB_MASTER_WriteRegs(&_MBChannel,&_aData[0],1000,2);  // Write Registers
}
```


4.1.4 Slave specific core function

4.1.4.1 MB_SLAVE_AddASCIIChannel()

Description

Adds a slave channel that uses the Modbus ASCII protocol via a serial connection.

Prototype

```
void MB_SLAVE_AddASCIIChannel(      MB_CHANNEL           * pChannel,
                                    MB_IFACE_CONFIG_UART * pConfig,
                              const MB_SLAVE_API         * pSlaveAPI,
                              const MB_IFACE_UART_API    * pIFaceAPI,
                                    U8                     SlaveAddr,
                                    U8                     DisableWrite,
                                    U32                    Baudrate,
                                    U8                     DataBits,
                                    U8                     Parity,
                                    U8                     StopBits,
                                    U8                     Port);
```

Parameters

Parameter	Description
pChannel	Pointer to element of MB_CHANNEL that is added to linked list.
pConfig	Pointer to element of MB_IFACE_CONFIG_UART.
pSlaveAPI	Pointer to element of MB_SLAVE_API used to read/write from/to target.
pIFaceAPI	Pointer to element of MB_IFACE_UART_API used to read/write from/to interface.
SlaveAddr	Slave addr. to listen on.
DisableWrite	Disable write access on this channel.
Baudrate	Desired baudrate to configure.
DataBits	Number of data bits used in UART protocol.
Parity	Parity used in UART protocol.
StopBits	Number of stop bits used in UART protocol.
Port	UART port number used by this channel.

Example

```
//
// Static declarations
//
static       MB_CHANNEL             _MBChannel;
static       MB_IFACE_CONFIG_UART   _MBConfig;
static const MB_IFACE_UART_API      _IFaceAPI = {
  _SendByte, _Init, _DeInit, NULL, NULL, NULL, NULL, NULL, NULL
};
static const MB_SLAVE_API           _SlaveAPI = {
  _WriteCoil, _ReadCoil, _ReadDI, _WriteReg, _ReadHR, _ReadIR
};

//
// Code running in main task
//
void MainTask(void) {
  MB_SLAVE_Init();                   // Init slave
  MB_SLAVE_AddASCIIChannel(&_MBChannel, &_MBConfig, &_SlaveAPI, &_IFaceAPI, 1, 0,
               38400, 8, 0, 1, 0);  // Add slave channel
  OS_CREATETASK( ... );              // Start slave task
}
```


4.1.4.2 MB_SLAVE_AddIPChannel()

Description

Adds a slave channel that uses the Modbus/TCP or Modbus/UDP protocol via IP.

Prototype

```
void MB_SLAVE_AddIPChannel(      MB_CHANNEL          * pChannel,
                                 MB_IFACE_CONFIG_IP  * pConfig,
                           const MB_SLAVE_API        * pSlaveAPI,
                           const MB_IFACE_IP_API     * pIFaceAPI,
                                 U8                    SlaveAddr,
                                 U8                    DisableWrite,
                                 U32                   IPAddr,
                                 U16                   Port);
```

Parameters

Parameter	Description
pChannel	Pointer to element of MB_CHANNEL that is added to linked list.
pConfig	Pointer to element of MB_IFACE_CONFIG_IP.
pSlaveAPI	Pointer to element of MB_SLAVE_API used to read/write from/to target.
pIFaceAPI	Pointer to element of MB_IFACE_IP_API used to read/write from/to interface.
SlaveAddr	Slave addr. to listen on.
DisableWrite	Disable write access on this channel.
IPAddr	Filter addr. If set, only connections on this addr. should be accepted.
Port	Port that accepts connections for this channel.

Example

```
//
// Static declarations
//
static       MB_CHANNEL          _MBChannel;
static       MB_IFACE_CONFIG_IP  _MBConfig;
static const MB_IFACE_IP_API     _IFaceAPI = {
  NULL, _Init, _DeInit, _Send, _Recv, _Connect, _Disconnect, NULL, NULL
};
static const MB_SLAVE_API          _SlaveAPI = {
  _WriteCoil, _ReadCoil, _ReadDI, _WriteReg, _ReadHR, _ReadIR
};

//
// Code running in main task
//
void MainTask(void) {
  MB_SLAVE_Init();          // Init slave
  OS_CREATETASK( ... );     // Start slave task
  MB_SLAVE_AddIPChannel(&_MBChannel, &_MBConfig, &_SlaveAPI, &_IFaceAPI, 1, 0,
             0, 502);       // Add slave channel
  OS_CREATETASK( ... );     // Start polling task for this channel
}
```

4.1.4.3 MB_SLAVE_AddRTUChannel()

Description

Adds a slave channel that uses the Modbus RTU protocol via a serial connection.

Prototype

```
void MB_SLAVE_AddRTUChannel(      MB_CHANNEL           * pChannel,
                                  MB_IFACE_CONFIG_UART * pConfig,
                            const MB_SLAVE_API         * pSlaveAPI,
                            const MB_IFACE_UART_API    * pIFaceAPI,
                                  U8                     SlaveAddr,
                                  U8                     DisableWrite,
                                  U32                    Baudrate,
                                  U8                     DataBits,
                                  U8                     Parity,
                                  U8                     StopBits,
                                  U8                     Port);
```

Parameters

Parameter	Description
pChannel	Pointer to element of MB_CHANNEL that is added to linked list.
pConfig	Pointer to element of MB_IFACE_CONFIG_UART.
pSlaveAPI	Pointer to element of MB_SLAVE_API used to read/write from/to target.
pIFaceAPI	Pointer to element of MB_IFACE_UART_API used to read/write from/to interface.
SlaveAddr	Slave addr. to listen on.
DisableWrite	Disable write access on this channel.
Baudrate	Desired baudrate to configure.
DataBits	Number of data bits used in UART protocol.
Parity	Parity used in UART protocol.
StopBits	Number of stop bits used in UART protocol.
Port	UART port number used by this channel.

Example

```
//
// Static declarations
//
static       MB_CHANNEL           _MBChannel;
static       MB_IFACE_CONFIG_UART _MBConfig;
static const MB_IFACE_UART_API    _IFaceAPI = {
  _SendByte, _Init, _DeInit, NULL, NULL, NULL, NULL, _InitTimer, _DeInitTimer
};
static const MB_SLAVE_API         _SlaveAPI = {
  _WriteCoil, _ReadCoil, _ReadDI, _WriteReg, _ReadHR, _ReadIR
};

//
// Code running in main task
//
void MainTask(void) {
  MB_SLAVE_Init();                 // Init slave
  MB_SLAVE_AddRTUChannel(&_MBChannel, &_MBConfig, &_SlaveAPI, &_IFaceAPI, 1, 0,
              38400, 8, 0, 1, 0);  // Add slave channel
  OS_CREATETASK( ... );            // Start slave task}
}
```


4.1.4.4 MB_SLAVE_ConfigIgnoreSlaveAddr()

Description

Function that configures if the slave address is ignored.

Prototype

```
void MB_SLAVE_ConfigIgnoreSlaveAddr(MB_CHANNEL * pChannel,
                                    U8           OnOff);
```

Parameters

Parameter	Description
pChannel	Pointer to element of MB_CHANNEL.
OnOff	Ignore the slave addr. and therefore allow slave addr. 0 to be used as a regular addr. • 0: Slave addr. 0 is treated as broadcast addr. • 1: Slave addr. is not checked, 0 is treated as a regular addr.

Additional information

For Modbus/TCP the standard expects a slave addr. of 0xFF to be used. However it is recommended to also accept the broadcast address 0 as it is an unicast connection for TCP. The same does not necessarily apply for UDP connections as well as they can actually be used for broadcast messages. As IP channels are basically unique by their used ports, the check for the slave ID can be disabled to allow a master to access a slave by simply knowing its port.

Example

```
//
// Static declarations
//
static       MB_CHANNEL         _MBChannel;
static       MB_IFACE_CONFIG_IP _MBConfig;
static const MB_IFACE_IP_API    _IFaceAPI = {
  NULL, _Init, _DeInit, _Send, _Recv, _Connect, _Disconnect, NULL, NULL
};
static const MB_SLAVE_API         _SlaveAPI = {
  _WriteCoil, _ReadCoil, _ReadDI, _WriteReg, _ReadHR, _ReadIR
};

//
// Code running in main task
//
void MainTask(void) {
  MB_SLAVE_Init();         // Init slave
  OS_CREATETASK( ... );    // Start slave task
  MB_SLAVE_AddIPChannel(&_MBChannel, &_MBConfig, &_SlaveAPI, &_IFaceAPI, 0, 0,
             0, 502);      // Add slave channel with address 0.
  MB_SLAVE_ConfigIgnoreSlaveAddr(&_MBChannel, 1);
  // Allow accepting connections for address 0.
  OS_CREATETASK( ... );    // Start polling task for this channel
}
```

4.1.4.5 MB_SLAVE_DeInit()

Description

De-Initializes the slave channels, resources and removes the slave endpoint from the global endpoint list.

Prototype

```
void MB_SLAVE_DeInit(void);
```

4.1.4.6 MB_SLAVE_Exec()

Description

Loops once over all slave channels and processes data when the channel has been signalled ready.

Prototype

```
void MB_SLAVE_Exec(void);
```

4.1.4.7 MB_SLAVE_Init()

Description

Initializes the slave resources and adds the slave endpoint to the global endpoint list.

Prototype

```
void MB_SLAVE_Init(void);
```


4.1.4.8 MB_SLAVE_PollChannel()

Description

Function that has to be periodically polled for each slave channel that requires requesting data instead of getting it via interrupt.

Prototype

```
int MB_SLAVE_PollChannel(MB_CHANNEL * pChannel);
```

Parameters

Parameter	Description
pChannel	Pointer to element of MB_CHANNEL.

Return value

< 0	Error.
= 0	No complete Modbus message signalled.
= 1	Complete Modbus message signalled.

Example

```
//
// Polling a slave channel in a task, allowing it to sleep when possible.
//
static void _PollSlaveChannelTask(void *pChannel) {
  while (1) {
    MB_SLAVE_PollChannel((MB_CHANNEL*)pChannel);
  }
}
```

4.1.4.9 MB_SLAVE_SetCustomFunctionCodeHandler()

Description

Sets a handler for custom function codes.

Prototype

```
void MB_SLAVE_SetCustomFunctionCodeHandler
                                             (MB_CHANNEL                       * pChannel,
                                              MB_pfCustomFunctionCodeHandler    pf);
```

Parameters

Parameter	Description
pChannel	Pointer to element of MB_CHANNEL.
pf	Callback function to execute.

Additional information

The callback is of type

```
int (*pf)( struct MB_CHANNEL                * pChannel,
                  MB_CUSTOM_FUNC_CODE_PARA * pPara);
```

with the following parameters.

Parameter	Description
pChannel	Context of type MB_CHANNEL provided to the callback.
pPara	Structure of parameters of type MB_CUSTOM_FUNC_CODE_PARA provided to the callback.

Example

```
/*********************************************************************
*
*       _HandleCustomFunctionCode()
*
* Function description
*   Handles custom function codes or overwrites stack internal handling.
*
* Parameters
*   pChannel: Pointer to element of type MB_CHANNEL.
*   pPara   : Input/output parameter structure.
*
* Return value
*   >= 0              : O.K., function code handled. Number of bytes to send back.
*   <  0              : Error, use official Modbus error codes
*                       like MB_ERR_ILLEGAL_DATA_VAL .
*   == MB_ERR_FUNC_CODE: Function code not handled. Try stack internal handling.
*
* Additional information
*   Function code 0x08 (Diagnostic), subfunction code 0x00 0x00 (Return Query Data) :
*     This very basic diagnostic subfunction echoes back a two byte
*     value that has just been received.
*   Function code 0x30 :
*     The payload of the message is expected to be a printable
*     string with termination. As the string itself is properly
*     terminated no length field is necessary.
*     One U8 is expected as return code that lets the master
*     know if the string has been printed or not. In this sample
*     this is decided by checking if MB_DEBUG is active or not.
*/
static int _HandleCustomFunctionCode(MB_CHANNEL               *pChannel,
                                     MB_CUSTOM_FUNC_CODE_PARA *pPara) {
  U32 SubCode;
  int r;
```



```
  MB_USE_PARA(pChannel);
  r = MB_ERR_FUNC_CODE;  // Assume that we can not handle this function code.

  //
  // Handle custom function codes.
  //
  switch (pPara->Function) {
  case 0x08:
    SubCode = MB_LoadU16BE((const U8*)pPara->pData);
    switch (SubCode) {
    case 0x0000:
      r = 4;  // Send back Subfunction Hi/Lo & Data Hi/Lo fields. Data is echoed
              // back as it is in the input/output buffer.
      break;
    }
    break;
  case 0x30:
    //
    // Output the string that has been sent.
    //
    MB_Log((const char*)pPara->pData);
    //
    // Store MB_DEBUG level as 1 byte answer.
    // Up to pPara->BufferSize bytes might be stored.
    //
    *pPara->pData  = MB_DEBUG;
    r              = 1;  // Tell the stack that we are sending back 1 byte data.
    break;
  }
  return r;
}

/*********************************************************************
*
*       MainTask()
*
* Function description
*   Main task executed by the RTOS to create further resources and
*   running the main application.
*/
void MainTask(void) {
  ...
  //
  // Start Modbus slave using Modbus/TCP protocol.
  //
  MB_SLAVE_Init();
  MB_SLAVE_AddIPChannel(&_MBChannel, ...;  // Add a slave channel.
  //
  // Add a custom function code handler for this channel.
  //
  MB_SLAVE_SetCustomFunctionCodeHandler(&_MBChannel, _HandleCustomFunctionCode);
  ...
}
```

4.1.4.10 MB_SLAVE_Task()

Description

Wrapper function that runs `MB_SLAVE_Exec()` in a task.

Prototype

```
void MB_SLAVE_Task(void);
```


4.1.5 Other core functions

4.1.5.1 MB_ConfigTimerFreq()

Description

This function allows setting a user defined timer frequency instead of the default 1kHz frequency.

Prototype

```
void MB_ConfigTimerFreq(U32 Freq);
```

Parameters

Parameter	Description
Freq	Timer frequency that shall be used for all all channels to calculate the RTU timeout.

4.1.5.2 MB_OnRx()

Description

Function called by byte oriented transmission channels that receive an interrupt for new data received.

Prototype

```
void MB_OnRx(MB_CHANNEL * pChannel,
             U8           Data);
```

Parameters

Parameter	Description
pChannel	Pointer to element of MB_CHANNEL.
Data	Received character.

4.1.5.3 MB_OnTx()

Description

Function called by byte oriented transmission channels once a Tx complete interrupt has been received to send the next byte or report back that there is no more to send.

Prototype

```
int MB_OnTx(MB_CHANNEL * pChannel);
```

Parameters

Parameter	Description
pChannel	Pointer to element of MB_CHANNEL.

Return value

< 0 Error.
= 0 More data sent.
= 1 No more data to send.

4.1.5.4 MB_TimerTick()

Description

Function called on each timer interrupt to manage internal RTU timeout with serial channels using the RTU protocol. Needs to be called by the user application each millisecond.

Prototype

```
void MB_TimerTick(void);
```

4.1.6 Helper functions

4.1.6.1 MB_LoadU16BE()

Description

Loads an U16 value from a big endian memory location.

Prototype

```
U32 MB_LoadU16BE(const U8 * pData);
```

Parameters

Parameter	Description
pData	Pointer to data location.

Return value

Data in target endianess.

4.1.6.2 MB_StoreU16BE()

Description

Stores an U16 value to a big endian memory location.

Prototype

```
void MB_StoreU16BE(U8   * pData,
                   U16    v);
```

Parameters

Parameter	Description
pData	Pointer to data location.
v	Value to store.

4.2 emModbus data structures

4.2.1 Interface configuration structures

4.2.1.1 Structure MB_IFACE_CONFIG_IP

Description

This structure holds configurations for IP communications.

Prototype

```
typedef struct {
  MB_SOCKET Sock;
  MB_SOCKET ListenSock;
  U32       IPAddr;
  U16       Port;
  U16       xID;
} MB_IFACE_CONFIG_API;
```

Member	Description
Sock	Socket used for send and receive.
ListenSock	Socket used by TCP for listen() and accept(). Not needed for UDP.
IPAddr	Master: Addr. to connect to. Slave: Filter address. If set only connections on this address should be accepted.
Port	Master: Port to connect to. Slave: Port that accepts connections for this channel.
xID	Master: Transaction ID that is incremented for each send. Slave: Ignored.

4.2.1.2 Structure MB_IFACE_CONFIG_UART

Description

This structure holds configurations for UART communications.

Prototype

```
typedef struct {
  U32 Cnt;
  U32 CntReload;
  U32 Baudrate;
  U8  DataBits;
  U8  Parity;
  U8  StopBits;
  U8  Port;
} MB_IFACE_CONFIG_UART;
```

Member	Description
Cnt	RTU timeout countdown.
CntReload	RTU countdown reload value.
Baudrate	Baudrate to use.
DataBits	Number of data bits.
Parity	Parity as interpreted by application.
StopBits	Number of stop bits.
Port	Interface index.

Additional information

MB_IFACE_CONFIG is of type MB_IFACE_CONFIG_UART .

4.2.2 Interface function structures

4.2.2.1 Structure MB_IFACE_IP_API

Description

This structure holds function pointers for IP communications.

Prototype

```
typedef struct {
  void ( *pfSendByte )    ( MB_IFACE_CONFIG_UART *pConfig,
                            U8                    Data );
  int  ( *pfInit )        ( MB_IFACE_CONFIG_UART *pConfig );
  void ( *pfDeInit )      ( MB_IFACE_CONFIG_UART *pConfig );
  int  ( *pfSend )        ( MB_IFACE_CONFIG_UART *pConfig,
                      const U8                   *pData,
                            U32                   NumBytes );
  int  ( *pfRecv )        ( MB_IFACE_CONFIG_UART *pConfig,
                            U8                   *pData,
                            U32                   NumBytes,
                            U32                   Timeout );
  int  ( *pfConnect )     ( MB_IFACE_CONFIG_UART *pConfig,
                            U32                   Timeout );
  void ( *pfDisconnect )  ( MB_IFACE_CONFIG_UART *pConfig );
  void ( *pfInitTimer )   ( U32                   MaxFreq );
  void ( *pfDeInitTimer ) ( void );
} MB_IFACE_IP_API;
```

Member	Description
pfSendByte	Send first byte. Every next byte will be sent via MB_OnTx() from interrupt. NULL if stream oriented interface, as pfSend gets used instead.
pfInit	Init IP and get listen socket and bring it in listen state if needed. NULL if not needed.
pfDeInit	Close listen socket and de-init IP. NULL if not needed.
pfSend	Send data for stream oriented interface. NULL if byte oriented interface is used, as pfSendByte gets used instead.
pfRecv	Request more data.
pfConnect	Master: Connect to slave. Slave: Accept connection if needed. NULL if not needed.
pfDisconnect	Master: Disconnect from slave. Slave: Close connection if needed. NULL if not needed.
pfInitTimer	NULL.
pfDeInitTimer	NULL.

4.2.2.2 Structure MB_IFACE_UART_API

Description

This structure holds function pointers for UART communications.

Prototype

```
typedef struct {
  void ( *pfSendByte )     ( MB_IFACE_CONFIG_UART *pConfig,
                             U8                    Data );
  int  ( *pfInit )         ( MB_IFACE_CONFIG_UART *pConfig );
  void ( *pfDeInit )       ( MB_IFACE_CONFIG_UART *pConfig );
  int  ( *pfSend )         ( MB_IFACE_CONFIG_UART *pConfig,
                       const U8                   *pData,
                             U32                   NumBytes );
  int  ( *pfRecv )         ( MB_IFACE_CONFIG_UART *pConfig,
                             U8                   *pData,
                             U32                   NumBytes,
                             U32                   Timeout );
  int  ( *pfConnect )      ( MB_IFACE_CONFIG_UART *pConfig,
                             U32                   Timeout );
  void ( *pfDisconnect )   ( MB_IFACE_CONFIG_UART *pConfig );
  void ( *pfInitTimer )    ( U32                   MaxFreq );
  void ( *pfDeInitTimer )  ( void );
} MB_IFACE_UART_API;
```

Member	Description
pfSendByte	Send first byte. Every next byte will be sent via MB_OnTx() from interrupt. NULL if stream oriented interface, as pfSend gets used instead.
pfInit	Init hardware. NULL if not needed.
pfDeInit	De-Init hardware. NULL if not needed.
pfSend	Send data for stream oriented interface. NULL if byte oriented interface, as pfSendByte gets used instead.
pfRecv	Typically data is received via MB_OnRx() from interrupt. NULL if not using polling mode.
pfConnect	NULL.
pfDisconnect	NULL.
pfInitTimer	Typically needed for RTU interfaces only. Initializes a timer needed for RTU timeout. NULL if not needed.
pfDeInitTimer	De-initialize RTU timer. NULL if not needed.

Additional information

MB_IFACE_API is of type MB_IFACE_UART_API.

4.2.3 Slave structures

4.2.3.1 Structure MB_SLAVE_API

Description

This structure holds function pointers used by slaves.

Prototype

```
typedef struct {
  int ( *pfWriteCoil ) ( U16 Addr, char OnOff );
  int ( *pfReadCoil )  ( U16 Addr );
  int ( *pfReadDI )    ( U16 Addr );
  int ( *pfWriteReg )  ( U16 Addr, U16   Val  );
  int ( *pfReadHR )    ( U16 Addr, U16  *pVal );
  int ( *pfReadIR )    ( U16 Addr, U16  *pVal );
} MB_SLAVE_API;
```

Member	Description
pfReadCoil	Read coil status.
pfReadDI	Read discrete input registers.
pfReadHR	Read holding register.
pfReadIR	Read input register.
pfWriteCoil	Write coil.
pfWriteReg	Write register.

4.2.3.2 Structure MB_CUSTOM_FUNC_CODE_PARA

Description

This structure holds function pointers used by slaves.

Prototype

```
typedef struct {
  U8   *pData;
  U32  DataLen;
  U32  BufferSize;
  U8   SlaveAddr;
  U8   Function;
} MB_CUSTOM_FUNC_CODE_PARA;
```

Member	Description
pData	Beginning of input/output buffer.
DataLen	Data length received.
BufferSize	Max. buffer size that can be used for an answer.
SlaveAddr	Slave addr. for which the message has been received.
Function	Function code received.

4.3 Error codes

The following table contains a list of emModbus error codes.

Generally, success is indicated by 0 and definite errors are indicated by negative numbers.

Symbolic name	Value	Description
Slave errors		
MB_ERR_ILLEGAL_FUNC	-1	The function code received in the query is an illegal function code for the slave. This may be because the function code was not implemented in the selected device. It could also indicate that the slave is in the wrong state to process a request of this type.
MB_ERR_ILLEGAL_DATA_ADDR	-2	The data address received in the query is an invalid address for the slave. More specifically, the combination of reference number and transfer length is invalid.
MB_ERR_ILLEGAL_DATA_VAL	-3	A value contained in the query data field is an invalid value for the slave. This indicates a fault in the structure of a request, such as an incorrect implied length.
MB_ERR_SLAVE_FAIL	-4	An unrecoverable error occurred while the slave was attempting to perform the requested action.
MB_ERR_ACK	-5	The slave has accepted the request and is processing it, but a long duration of time will be required to do so. This response is returned to prevent a timeout error from occurring in the master, which can then poll for process completion.
MB_ERR_SLAVE_BUSY	-6	The slave is engaged in processing a long-duration command. The master should retransmit the message later.
MB_ERR_NACK	-7	The requested function cannot be performed. Issue poll to obtain detailed device dependent error information.
MB_ERR_MEM_PARITY_ERR	-8	The slave attempted to perform the query, but detected a parity error in the memory. The master can retry the request, but service may be required on the slave device.
Stack internal errors		
MB_ERR_MISC	-20	Unspecified error.
MB_ERR_CONNECT	-21	Error while connecting.
MB_ERR_CONNECT_TIMEOUT	-22	Timeout while connecting.
MB_ERR_DISCONNECT	-23	Interface signaled disconnect.
MB_ERR_TIMEOUT	-24	No answer received on request.
MB_ERR_CHECKSUM	-25	Received message did not pass LRC/CRC check.
MB_ERR_PARAM	-26	Parameter error in API call.
MB_ERR_SLAVE_ADDR	-27	Received valid response with wrong slave address.

Symbolic name	Value	Description
MB_ERR_FUNC_CODE	-28	Received valid response with wrong function code.
MB_ERR_REF_NO	-29	Received valid response with wrong reference number.
MB_ERR_NUM_ITEMS	-30	Received valid response with more or less items than requested.
MB_ERR_DATA	-31	Received valid response for a single write with different data than written.
MB_ERR_TRIAL_LIMIT	-32	Trial limit exceeded. When using trial libraries, this error occurs after 12 hours of run time. Except from this, the trial library is fully functional and includes all features of emModbus.
MB_ERR_WOULD_BLOCK	-33	TCP non-blocking recv() would block situation.

Chapter 5

Configuring emModbus

emModbus can be used without changing any of the compile-time flags. All compile-time configuration flags are preconfigured with valid values, which match the requirements of most applications.

The default configuration of emModbus can be changed via compile-time flags which can be added to `MB_Conf.h`. `MB_Conf.h` is the main configuration file for the emModbus stack.

5.1 Compile-time configuration

The following types of configuration macros exist:

Binary switches "B"

Switches can have a value of either 0 or 1, for deactivated and activated respectively. Actually, anything other than 0 works, but 1 makes it easier to read a configuration file. These switches can enable or disable a certain functionality or behavior. Switches are the simplest form of configuration macros.

Numerical values "N"

Numerical values are used somewhere in the code in place of a numerical constant. A typical example is the configuration of the sector size of a storage medium.

Function replacements "F"

Macros can be treated like regular functions although certain limitations apply, as a macro is still put into the code as simple text replacement. Function replacements are mainly used to add specific functionality to a module which is highly hardware dependent. This type of macro is always declared using brackets (and optional parameters).

5.1.1 Compile-time configuration switches

Type	Symbolic name	Default	Description
System configuration macros			
B	MB_IS_BIGENDIAN	0	Macro to define if a big endian target is used.
B	MB_ALLOW_STREAM_HDR_UNDERFLOW	0	Macro to allow to receive the header on a streaming interface (Modbus/TCP or Modbus/UDP) in multiple recv() calls. Typically this should never happen and if it happens this typically means that communication got out of sync.
B	MB_DISCONNECT_ON_MSG_TOO_BIG	1	Macro to define if we disconnect if a message bigger than MB_MAX_MESSAGE_SIZE is received (determined by header length field).
Debug macros			
N	MB_DEBUG	0	Macro to define the debug level of the emModbus build. Refer to *Debug level* on page 89 for a description of the different debug level.
Optimization macros			
F	MB_MEMCMP	memcmp (C-routine in standard C-library)	Macro to define an optimized memcmp routine to speed up the stack. An optimized memcmp routine is typically implemented in assembly language.
F	MB_MEMCPY	memcpy (C-routine in standard C-library)	Macro to define an optimized memcpy routine to speed up the stack. An optimized memcpy routine is typically implemented in assembly language. Optimized versions for IAR and GCC compilers are supplied.

Type	Symbolic name	Default	Description
F	MB_MEMMOVE	memmove (C-routine in standard C-library)	Macro to define an optimized memmove routine to speed up the stack. An optimized memmove routine is typically implemented in assembly language.
F	MB_MEMSET	memset (C-routine in standard C-library)	Macro to define an optimized memset routine to speed up the stack. An optimized memset routine is typically implemented in assembly language.

5.1.2 Debug level

emModbus can be configured to display debug information at higher debug levels to locate a problem (Error) or potential problem. To display information, emModbus uses the logging routines (see chapter *Debugging* on page 91). These routines can be blank, they are not required for the functionality of emModbus. In a target system, they are typically not required in a release (production) build, since a production build typically uses a lower debug level.

If (`IP_DEBUG = 0`): used for release builds. Includes no debug options.
If (`IP_DEBUG = 1`): `MP_PANIC()` is mapped to `MP_Panic()`.
If (`IP_DEBUG ≥ 2`): `MP_PANIC()` is mapped to `MP_Panic()` and logging support is activated.

Chapter 6

Debugging

emModbus comes with debugging options including optional warning and log outputs.

6.1 Message output

The debug builds of emModbus include a debug system which helps to analyze the correct implementation of the stack in your application. All modules can output logging and warning messages via terminal I/O.

6.1.1 Debug API functions

Function	Description
I/O functions	
MB_Log()	This function is called by the stack in debug builds with log output.
MB_Panic()	This function is called if the stack encounters a critical situation.
MB_Warn()	This function is called by the stack in debug builds with warning output.

6.1.1.1 MB_Log()

Description

This function is called by the stack in debug builds with log output. In a release build, this function may not be linked in.

Prototype

```
void MB_Log(const char * s);
```

Parameters

Parameter	Description
s	String to output.

Additional information

Interrupts and task switches: printf() has a re-entrance problem on a lot of systems if interrupts are not disabled. Strings to output would be scrambled if during an output from a task an output from an interrupt would take place. In order to avoid this problem, interrupts are disabled.

6.1.1.2 MB_Panic()

Description

This function is called if the stack encounters a critical situation. In a release build, this function may not be linked in.

Prototype

```
void MB_Panic(const char * s);
```

Parameters

Parameter	Description
s	String to output.

6.1.1.3 MB_Warn()

Description

This function is called by the stack in debug builds with warning output. In a release build, this function may not be linked in.

Prototype

```
void MB_Warn(const char * s);
```

Parameters

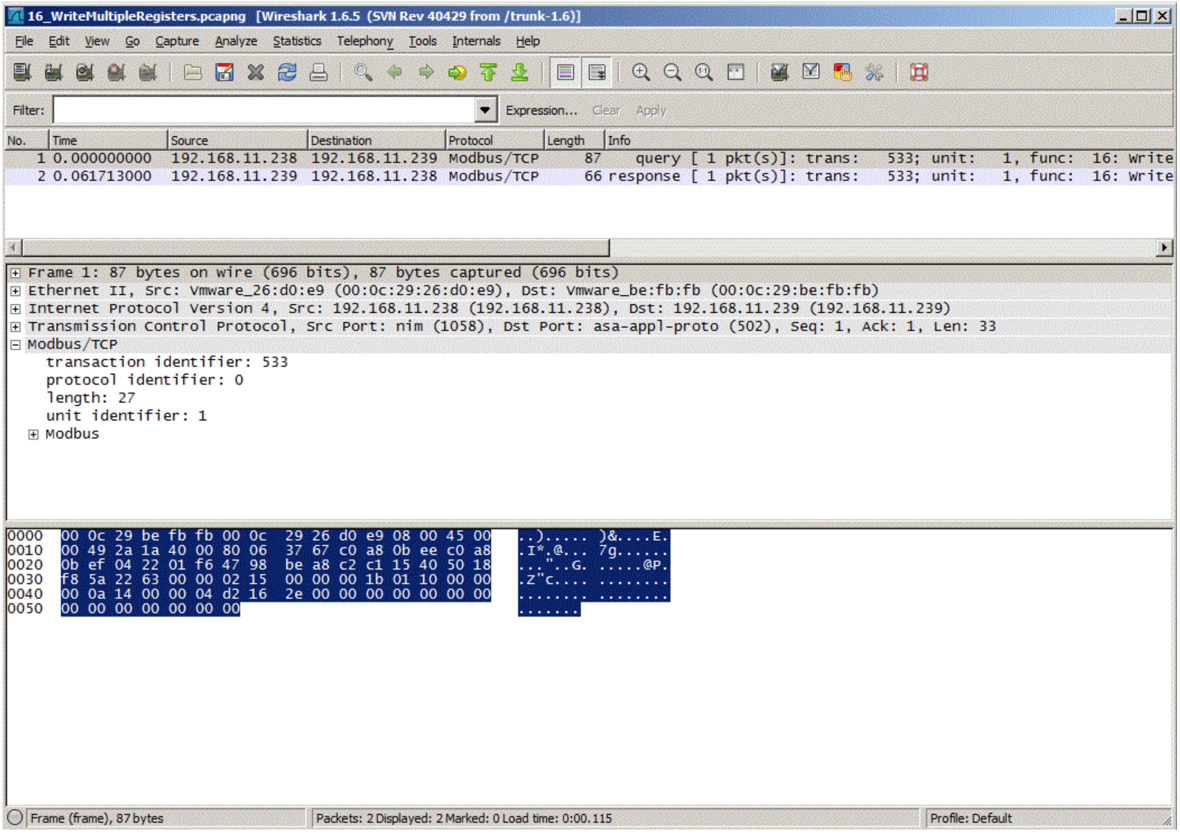
Parameter	Description
s	String to output.

Additional information

Interrupts and task switches: printf() has a re-entrance problem on a lot of systems if interrupts are not disabled. Strings to output would be scrambled if during an output from a task an output from an interrupt would take place. In order to avoid this problem, interrupts are disabled.

6.2 Using a network sniffer to analyse Ethernet communication problems

Using a network sniffer to analyze your local Ethernet traffic may give you a deeper understanding of the data that is being sent in your network. For this purpose you can use several network sniffers. Some of them are available for free such as *Wireshark*. An example of a network sniff using *Wireshark* is shown in the screenshot below:



6.3 Testing emModbus applications

We recommend testing emModbus devices by using their respective counterparts, e.g. using a emModbus/TCP master to test a emModbus/TCP slave and vice versa. Alternatively, devices can also be tested with a desktop computer running an appropriate Modbus application.

To solely test emModbus on target hardware, we recommend building a corresponding project for the specific application. For example, the application contained in `OS_IP_MB_SlaveTCP.c`, can be tested using a project for the application contained in `OS_IP_MB_MasterTCP.c`. Configuration of some parameters (e.g. IP address) is required before compiling the project and downloading the output into a second target. When connected to the same network, both devices should then start communication with each other.

To test emModbus using a desktop computer, an appropriate software package is required. The shipment contains Windows applications for Modbus master and slave devices using Modbus/TCP, which can be used to test both devices via that connection. In addition, several vendors offer Modbus testing applications for Microsoft Windows and other operating systems, many of which are free or at least free to evaluate for a limited time. We recommend "Modbus Poll" for testing emModbus slave functionalities and "Modbus Slave" for testing emModbus master functionalities. Both applications can be downloaded from http://www.modbustools.com.

Chapter 7

OS Integration

emModbus is designed to be used in a multitasking environment. The interface to the operating system is encapsulated in a single file, the MB/OS interface. For embOS, all functions required for this MB/OS interface are implemented in a single file which comes with emModbus.

This chapter provides descriptions of the functions required to fully support emModbus in multitasking environments.

7.1 General information

All OS interface functions for embOS are implemented in MB_X_embOS.c, which is located in the Shared folder of the emModbus stack.

7.2 OS layer API functions

Function	Description
General functions	
MB_OS_DeInitMaster()	De-Initialize (remove) all objects required for task syncronisation of the master.
MB_OS_DeInitSlave()	De-Initialize (remove) all objects required for task synchronisation and signalling of the slave.
MB_OS_DisableInterrupt()	Disables interrupts before critical operations.
MB_OS_EnableInterrupt()	Enables interrupts after critical operations.
MB_OS_GetTime()	Return the current system time in ms.
MB_OS_InitMaster()	Initialize (create) all objects required for task synchronisation of the master.
MB_OS_InitSlave()	Initialize (create) all objects required for task synchronisation and signalling of the slave.
Synchronization functions	
MB_OS_SignalItem()	Sets an object to signaled state, or resumes tasks which are waiting at the event object.
MB_OS_SignalNetEvent()	Wakes the MB_SLAVE_Task() waiting for a NET-event or timeout in the function MB_OS_WaitNetEvent().
MB_OS_WaitItemTimed()	Suspends a task which needs to wait for an object.
MB_OS_WaitNetEvent()	Blocks until the timeout expires or a NET-event occurs, meaning MB_SignalNetEvent() is called from an ISR.

7.2.1 General functions

7.2.1.1 MB_OS_DeInitMaster()

Description

De-Initialize (remove) all objects required for task syncronisation of the master. If the entire stack executes from a single task, no functionality is required here.

Prototype

```
void MB_OS_DeInitMaster(void);
```


7.2.1.2 MB_OS_DeInitSlave()

Description

De-Initialize (remove) all objects required for task synchronisation and signalling of the slave. If the entire stack executes from a single task, no functionality is required here.

Prototype

```
void MB_OS_DeInitSlave(void);
```

7.2.1.3 MB_OS_DisableInterrupt()

Description

Disables interrupts before critical operations.

Prototype

```
void MB_OS_DisableInterrupt(void);
```


7.2.1.4 MB_OS_EnableInterrupt()

Description

Enables interrupts after critical operations.

Prototype

```
void MB_OS_EnableInterrupt(void);
```


7.2.1.5 MB_OS_GetTime()

Description

Return the current system time in ms. The value will wrap around after approximately 49.7 days. This is taken into account by the stack.

Prototype

```
U32 MB_OS_GetTime(void);
```

Return value

System time in ms.

7.2.1.6 MB_OS_InitMaster()

Description

Initialize (create) all objects required for task synchronisation of the master. This is one semaphore for protection of critical code, which may not be executed from multiple tasks at the same time, and a hook in case a task currently executing Modbus master API is terminated.

Prototype

```
void MB_OS_InitMaster(void);
```


7.2.1.7 MB_OS_InitSlave()

Description

Initialize (create) all objects required for task synchronisation and signalling of the slave. This is one semaphore for protection of critical code which may not be executed from multiple task at the same time.

Prototype

```
void MB_OS_InitSlave(void);
```


7.2.2 Synchronization functions

7.2.2.1 MB_OS_SignalItem()

Description

Sets an object to signaled state, or resumes tasks which are waiting at the event object.

Prototype

```
void MB_OS_SignalItem(void * pWaitItem);
```

Parameters

Parameter	Description
pWaitItem	Pointer to item a task is waiting for.

7.2.2.2 MB_OS_SignalNetEvent()

Description

Wakes the MB_SLAVE_Task() waiting for a NET-event or timeout in the function MB_OS_WaitNetEvent(). If the entire stack executes from a single task, no functionality is required here.

Prototype

```
void MB_OS_SignalNetEvent(void);
```


7.2.2.3 MB_OS_WaitItemTimed()

Description

Suspends a task which needs to wait for an object. This object is identified by a pointer to it and can be of any type, e.g. channel.

Prototype

```
void MB_OS_WaitItemTimed(void     * pWaitItem,
                         unsigned   Timeout);
```

Parameters

Parameter	Description
pWaitItem	Pointer to item a task shall wait for until signalled or timeout occurs.
Timeout	Timeout [ms] to wait for item to be signalled.

7.2.2.4 MB_OS_WaitNetEvent()

Description

Blocks until the timeout expires or a NET-event occurs, meaning MB_SignalNetEvent() is called from an ISR. If the entire stack executes from a single task, no functionality is required here. Called from MB_SLAVE_Task() only.

Prototype

```
void MB_OS_WaitNetEvent(unsigned ms);
```

Parameters

Parameter	Description
ms	Time to wait for a NET-event to occur in ms. 0 for infinite.

Chapter 8

Resource usage

This chapter covers the resource usage of emModbus. It contains information about the memory requirements in typical systems, which can be used to obtain sufficient estimates for most target systems.

8.1 Memory footprint

emModbus is designed to fit many kinds of embedded design requirements. Some features might be excluded from a build to get a minimal system. Note that the values are only valid for the given configurations.

8.1.1 ARM7 system

The following table shows the hardware and the toolchain details of the project:

Detail	Description
CPU	ARM7
Tool chain	IAR Embedded Workbench for ARM V6.30.6
Model	ARM7, Thumb instructions; interwork;
Compiler options	Highest size optimization;

8.1.1.1 ROM usage

The following table shows the ROM requirement of emModbus:

Description	ROM
master using ASCII	approx. 1.5 Kbytes
master using TCP	approx. 0.9 Kbytes
master using RTU	approx. 2.1 Kbytes
slave using ASCII	approx. 2.0 Kbytes
slave using TCP	approx. 1.6 Kbytes
slave using RTU	approx. 2.6 Kbytes

8.1.1.2 RAM usage

emModus requires approximately 30 Bytes of RAM for the stack itself and approximately 300 Bytes of RAM for each channel added.

Chapter 9

Support

This chapter should help if any problem occurs, e.g. with the use of the emModbus functions, and describes how to contact the emModbus support.

9.1 Contacting support

If you are a registered emModbus user and you need to contact the emModbus sup- port, please send the following information via email to support@segger.com:

- Which emModbus do you use? (Master/Slave)
- The emModbus version.
- Your emModbus registration number.
- If you are unsure about the above information, you may also use the name of the emModbus ZIP-file (which contains the above information).
- A detailed description of the problem.
- Optionally, a project with which we can reproduce the problem.

Chapter 10

Index

