FreeModbus

1.1.0

Introduction

The latest version of this document is available on http://freemodbus.berlios.de/api.

FreeModbus is a Modbus ASCII/RTU and Modbus TCP implementation for embedded systems. It provides an implementation of the *Modbus Application Protocol v1.1a* and supports the RTU/ASCII transmission modes defined in the *Modbus over serial line specification 1.0*. Since version 0.7 FreeModbus also supports Modbus/TCP. Version 0.9 added the first Modbus/TCP port for embedded systems using the lwIP TCP/IP stack.

Ports

ARM devices:

- STR71X with FreeRTOS/GCC. See STR71X/simple2.c for an example.
- STR71TCP with FreeRTOS/lwIP/GCC. This port includes FreeRTOS, lwIP and a fully working PPP stack. The lwIP, PPP and FreeRTOS part is generic and therefore can be used for other ports (or other projects).

AVR devices:

- ATMega8/16/32/128/168/169 with WinAVR. See AVR/demo.c for an example.

Coldfire devices:

- MCF5235 with GCC. See MCF5235/demo.c for an example.
- MCF5235/TCP with GCC. This port features FreeRTOS and the lwIP stack. The lwIP part is generic and therefore it should be used as a basis for other lwIP ports.

MSP430 devices

- MSP430F169 with Rowley Crossworks. See MSP430/demo.c for an example.

- MSP430F169 with GCC. See MSP430/demo.c for an example.

Win32:

- A Win32 Modbus RTU/ASCII Port.
- A Win32 Modbus/TCP Port.

Linux:

- A Linux (uCLinux or other distributions) Modbus RTU/ASCII Port.

Automatically generated by Doxygen 1.3.9.1 on 19 Nov 2006.

FreeModbus Modules

Here is a list of all modules:

- **Modbus**
- **Modbus Registers**
- **Modbus Configuration**
- **Utilities**

Automatically generated by Doxygen 1.3.9.1 on 19 Nov 2006.

Modbus

Detailed Description

```
#include "mb.h"
```

This module defines the interface for the application. It contains the basic functions and types required to use the Modbus protocol stack. A typical application will want to call **eMBInit()** first. If the device is ready to answer network requests it must then call **eMBEnable()** to activate the protocol stack. In the main loop the function **eMBPoll()** must be called periodically. The time interval between pooling depends on the configured Modbus timeout. If an RTOS is available a separate task should be created and the task should always call the function **eMBPoll()**.

```
// Initialize protocol stack in RTU mode for a slave with address 10 = 0x0A
eMBInit( MB_RTU, 0x0A, 38400, MB_PAR_EVEN );
// Enable the Modbus Protocol Stack.
eMBEnable(  );
for( ;; )
{
    // Call the main polling loop of the Modbus protocol stack.
    eMBPoll(  );
    ...
}
```

Defines

#define **MB_TCP_PORT_USE_DEFAULT** 0

Enumerations

enum **eMBMode** { **MB_RTU**, **MB_ASCII**, **MB_TCP** }

enum **eMBRegisterMode** { **MB_REG_READ**, **MB_REG_WRITE** }

enum **eMBErrorCode** {
MB_ENOERR, **MB_ENOREG**, **MB_EINVAL**, **MB_EPORTERR**,
MB_ENORES, **MB_EIO**, **MB_EILLSTATE**, **MB_ETIMEDOUT**
}

enum **eMBParity** { **MB_PAR_NONE**, **MB_PAR_ODD**, **MB_PAR_EVEN** }

Functions

eMBErrorCode eMBInit (**eMBMode** eMode, UCHAR ucSlaveAddress, UCHAR ucPort, ULONG ulBaudRate, **eMBParity** eParity)

eMBErrorCode eMBTCPInit (USHORT usTCPPort)

eMBErrorCode eMBClose (void)

eMBErrorCode eMBEnable (void)

eMBErrorCode eMBDisable (void)

eMBErrorCode eMBPoll (void)

eMBErrorCode eMBSetSlaveID (UCHAR ucSlaveID, BOOL xIsRunning, UCHAR const *pucAdditional, USHORT usAdditionalLen)

eMBErrorCode eMBRegisterCB (UCHAR ucFunctionCode, pxMBFunctionHandler pxHandler)

Define Documentation

#define
MB_TCP_PORT_USE_DEFAULT 0

Use the default Modbus TCP port (502).

Enumeration Type Documentation

enum
eMBErrorCode

Errorcodes used by all function in the protocol stack.

Enumeration values:

MB_ENOERR	no error.
MB_ENOREG	illegal register address.
MB_EINVAL	illegal argument.
MB_EPORTERR	porting layer error.
MB_ENORES	insufficient resources.
MB_EIO	I/O error.
MB_EILLSTATE	protocol stack in illegal state.
MB_ETIMEDOUT	timeout error occurred.

enum
eMBMode

Modbus serial transmission modes (RTU/ASCII).

Modbus serial supports two transmission modes. Either ASCII or RTU. RTU is faster but has more hardware requirements and requires a network with a low jitter. ASCII is slower and more reliable on slower links (E.g. modems)

Enumeration values:

MB_RTU	RTU transmission mode.
MB_ASCII	ASCII transmission mode.
MB_TCP	TCP mode.

enum
eMBParity

Parity used for characters in serial mode.

The parity which should be applied to the characters sent over the serial link. Please note that this values are actually passed to the porting layer and therefore not all parity modes might be available.

Enumeration values:

MB_PAR_NONE	No parity.
MB_PAR_ODD	Odd parity.
MB_PAR_EVEN	Even parity.

enum
eMBRegisterMode

If register should be written or read.

This value is passed to the callback functions which support either reading or writing register values. Writing means that the application registers should be updated and reading means that the modbus protocol stack needs to know the current register values.

See also:
eMBRegHoldingCB( ), **eMBRegCoilsCB( )**, **eMBRegDiscreteCB( )** and **eMBRegInputCB( )**.

Enumeration values:

MB_REG_READ	Read register values and pass to protocol stack.
MB_REG_WRITE	Update register values.

Function Documentation

eMBErrorCode (void)
eMBClose

Release resources used by the protocol stack.

This function disables the Modbus protocol stack and release all hardware resources. It must only be called when the protocol stack is disabled.

Note:
Note all ports implement this function. A port which wants to get an callback must define the macro MB_PORT_HAS_CLOSE to 1.

Returns:
If the resources where released it return **eMBErrorCode::MB_ENOERR**. If the protocol stack is not in the disabled state it returns **eMBErrorCode::MB_EILLSTATE**.

Examples:
LINUX/demo.c, **MCF5235TCP/demo.c**, **STR71XTCP/demo.c**, **WIN32/demo.cpp**, and **WIN32TCP/demo.cpp**.

eMBErrorCode (void)
eMBDisable

Disable the Modbus protocol stack.

This function disables processing of Modbus frames.

Returns:
If the protocol stack has been disabled it returns **eMBErrorCode::MB_ENOERR**. If it was not in the enabled state it returns **eMBErrorCode::MB_EILLSTATE**.

Examples:
LINUX/demo.c, **MCF5235TCP/demo.c**, **STR71XTCP/demo.c**, **WIN32/demo.cpp**, and **WIN32TCP/demo.cpp**.

eMBErrorCode (void)
eMBEnable

Enable the Modbus protocol stack.

This function enables processing of Modbus frames. Enabling the protocol stack is only possible if it is in the disabled state.

Returns:

If the protocol stack is now in the state enabled it returns **eMBErrorCode::MB_ENOERR**. If it was not in the disabled state it return **eMBErrorCode::MB_EILLSTATE**.

Examples:

AVR/demo.c, **LINUX/demo.c**, **MCF5235/demo.c**, **MCF5235TCP/demo.c**, **MSP430/demo.c**, **STR71X/simple2.c**, **STR71XTCP/demo.c**, **WIN32/demo.cpp**, and **WIN32TCP/demo.cpp**.

```
eMBErrorCode eMBInit ( eMBMode eMode,
                       UCHAR ucSlaveAddress,
                       UCHAR ucPort,
                       ULONG ulBaudRate,
                       eMBParity eParity
                     )
```

Initialize the Modbus protocol stack.

This functions initializes the ASCII or RTU module and calls the init functions of the porting layer to prepare the hardware. Please note that the receiver is still disabled and no Modbus frames are processed until **eMBEnable( )** has been called.

Parameters:

eMode	If ASCII or RTU mode should be used.
ucSlaveAddress	The slave address. Only frames sent to this address or to the broadcast address are processed.
ucPort	The port to use. E.g. 1 for COM1 on windows. This value is platform dependent and some ports simply choose to ignore it.
ulBaudRate	The baudrate. E.g. 19200. Supported baudrates depend on the porting layer.
eParity	Parity used for serial transmission.

Returns:

If no error occurs the function returns **eMBErrorCode::MB_ENOERR**. The protocol is then in the disabled state and ready for activation by calling **eMBEnable( )**. Otherwise one of the following error codes is returned:

- **eMBErrorCode::MB_EINVAL** If the slave address was not valid. Valid slave addresses are in the range 1 - 247.
- **eMBErrorCode::MB_EPORTERR** IF the porting layer returned an error.

Examples:

AVR/demo.c, **LINUX/demo.c**, **MCF5235/demo.c**, **MSP430/demo.c**, **STR71X/simple2.c**, and **WIN32/demo.cpp**.

eMBErrorCode **(void)**
eMBPoll

The main pooling loop of the Modbus protocol stack.

This function must be called periodically. The timer interval required is given by the application dependent Modbus slave timeout. Internally the function calls xMBPortEventGet() and waits for an event from the receiver or transmitter state machines.

Returns:

If the protocol stack is not in the enabled state the function returns **eMBErrorCode::MB_EILLSTATE**. Otherwise it returns **eMBErrorCode::MB_ENOERR**.

Examples:

AVR/demo.c, **LINUX/demo.c**, **MCF5235/demo.c**, **MCF5235TCP/demo.c**, **MSP430/demo.c**, **STR71X/simple2.c**, **STR71XTCP/demo.c**, **WIN32/demo.cpp**, and **WIN32TCP/demo.cpp**.

eMBErrorCode **(UCHAR** ***ucFunctionCode*****,**
eMBRegisterCB
pxMBFunctionHandler ***pxHandler***
)

Registers a callback handler for a given function code.

This function registers a new callback handler for a given function code. The callback handler supplied is responsible for interpreting the Modbus PDU and the creation of an appropriate response. In case of an error it should return one of the possible Modbus exceptions which results in a Modbus exception frame sent by the protocol stack.

Parameters:

ucFunctionCode	The Modbus function code for which this handler should be registers. Valid function codes are in the range 1 to 127.
pxHandler	The function handler which should be called in case such a frame is received. If `NULL` a previously registered function handler for this function code is removed.

Returns:

eMBErrorCode::MB_ENOERR if the handler has been installed. If no more resources are available it returns **eMBErrorCode::MB_ENORES**. In this case the values in **mbconfig.h** should be adjusted. If the argument was not valid it returns **eMBErrorCode::MB_EINVAL**.

eMBErrorCode **eMBSetSlaveID** (**UCHAR** *ucSlaveID*, **BOOL** *xIsRunning*, **UCHAR const *** *pucAdditional*, **USHORT** *usAdditionalLen*)

Configure the slave id of the device.

This function should be called when the Modbus function *Report Slave ID* is enabled (By defining MB_FUNC_OTHER_REP_SLAVEID_ENABLED in **mbconfig.h**).

Parameters:

ucSlaveID	Values is returned in the *Slave ID* byte of the *Report Slave ID* response.
xIsRunning	If TRUE the *Run Indicator Status* byte is set to 0xFF. otherwise the *Run Indicator Status* is 0x00.
pucAdditional	Values which should be returned in the *Additional* bytes of the *Report Slave ID* response.
usAdditionalLen	Length of the buffer `pucAdditonal`.

Returns:

If the static buffer defined by MB_FUNC_OTHER_REP_SLAVEID_BUF in **mbconfig.h** is to small it returns **eMBErrorCode::MB_ENORES**. Otherwise it returns **eMBErrorCode::MB_ENOERR**.

Examples:

AVR/demo.c, **LINUX/demo.c**, **MCF5235/demo.c**, and **WIN32/demo.cpp**.

eMBErrorCode **eMBTCPInit** (**USHORT** *usTCPPort*)

Initialize the Modbus protocol stack for Modbus TCP.

This function initializes the Modbus TCP Module. Please note that frame processing is still disabled until **eMBEnable( )** is called.

Parameters:

usTCPPort	The TCP port to listen on.

Returns:

If the protocol stack has been initialized correctly the function returns **eMBErrorCode::MB_ENOERR**. Otherwise one of the following error codes is returned:

- **eMBErrorCode::MB_EINVAL** If the slave address was not valid. Valid slave addresses are in the range 1 - 247.
- **eMBErrorCode::MB_EPORTERR** IF the porting layer returned an error.

Examples:

MCF5235TCP/demo.c, **STR71XTCP/demo.c**, and **WIN32TCP/demo.cpp**.

Automatically generated by Doxygen 1.3.9.1 on 19 Nov 2006.

Modbus Registers

Detailed Description

```
#include "mb.h"
```

The protocol stack does not internally allocate any memory for the registers. This makes the protocol stack very small and also usable on low end targets. In addition the values don't have to be in the memory and could for example be stored in a flash.
Whenever the protocol stack requires a value it calls one of the callback function with the register address and the number of registers to read as an argument. The application should then read the actual register values (for example the ADC voltage) and should store the result in the supplied buffer. If the protocol stack wants to update a register value because a write register function was received a buffer with the new register values is passed to the callback function. The function should then use these values to update the application register values.

Functions

eMBErrorCode	**eMBRegInputCB** (UCHAR *pucRegBuffer, USHORT usAddress, USHORT usNRegs)
eMBErrorCode	**eMBRegHoldingCB** (UCHAR *pucRegBuffer, USHORT usAddress, USHORT usNRegs, **eMBRegisterMode** eMode)
eMBErrorCode	**eMBRegCoilsCB** (UCHAR *pucRegBuffer, USHORT usAddress, USHORT usNCoils, **eMBRegisterMode** eMode)
eMBErrorCode	**eMBRegDiscreteCB** (UCHAR *pucRegBuffer, USHORT usAddress, USHORT usNDiscrete)

Function Documentation

eMBErrorCode **eMBRegCoilsCB**	**(UCHAR ***	***pucRegBuffer*,**
	USHORT	***usAddress*,**
	USHORT	***usNCoils*,**
	eMBRegisterMode	***eMode***
	)	

Callback function used if a *Coil Register* value is read or written by the protocol stack. If you are going to use this function you might use the functions **xMBUtilSetBits( )** and **xMBUtilGetBits( )** for working with bitfields.

Parameters:

pucRegBuffer	The bits are packed in bytes where the first coil starting at address `usAddress` is stored in the LSB of the first byte in the buffer `pucRegBuffer`. If the buffer should be written by the callback function unused coil values (I.e. if not a multiple of eight coils is used) should be set to zero.
usAddress	The first coil number.
usNCoils	Number of coil values requested.
eMode	If **eMBRegisterMode::MB_REG_WRITE** the application values should be updated from the values supplied in the buffer `pucRegBuffer`. If **eMBRegisterMode::MB_REG_READ** the application should store the current values in the buffer `pucRegBuffer`.

Returns:

The function must return one of the following error codes:

- **eMBErrorCode::MB_ENOERR** If no error occurred. In this case a normal Modbus response is sent.
- **eMBErrorCode::MB_ENOREG** If the application does not map an coils within the requested address range. In this case a **ILLEGAL DATA ADDRESS** is sent as a response.
- **eMBErrorCode::MB_ETIMEDOUT** If the requested register block is currently not available and the application dependent response timeout would be violated. In this case a **SLAVE DEVICE BUSY** exception is sent as a response.
- **eMBErrorCode::MB_EIO** If an unrecoverable error occurred. In this case a **SLAVE DEVICE FAILURE** exception is sent as a response.

Examples:

AVR/demo.c, **LINUX/demo.c**, **MCF5235/demo.c**, **MCF5235TCP/demo.c**, **MSP430/demo.c**, **STR71X/simple2.c**, **STR71XTCP/demo.c**, **WIN32/demo.cpp**, and **WIN32TCP/demo.cpp**.

```
eMBErrorCode
eMBRegDiscreteCB
                    ( UCHAR *
                        pucRegBuffer,
                      USHORT usAddress,
                      USHORT usNDiscrete
                    )
```

Callback function used if a *Input Discrete Register* value is read by the protocol stack.

If you are going to use his function you might use the functions **xMBUtilSetBits( )** and **xMBUtilGetBits( )** for working with bitfields.

Parameters:

pucRegBuffer	The buffer should be updated with the current coil values. The first discrete input starting at `usAddress` must be stored at the LSB of the first byte in the buffer. If the requested number is not a multiple of eight the remaining bits should be set to zero.
usAddress	The starting address of the first discrete input.
usNDiscrete	Number of discrete input values.

Returns:

The function must return one of the following error codes:

- **eMBErrorCode::MB_ENOERR** If no error occurred. In this case a normal Modbus response is sent.
- **eMBErrorCode::MB_ENOREG** If no such discrete inputs exists. In this case a **ILLEGAL DATA ADDRESS** exception frame is sent as a response.
- **eMBErrorCode::MB_ETIMEDOUT** If the requested register block is currently not available and the application dependent response timeout would be violated. In this case a **SLAVE DEVICE BUSY** exception is sent as a response.
- **eMBErrorCode::MB_EIO** If an unrecoverable error occurred. In this case a **SLAVE DEVICE FAILURE** exception is sent as a response.

Examples:

AVR/demo.c, **LINUX/demo.c**, **MCF5235/demo.c**, **MCF5235TCP/demo.c**, **MSP430/demo.c**, **STR71X/simple2.c**, **STR71XTCP/demo.c**, **WIN32/demo.cpp**, and **WIN32TCP/demo.cpp**.

```
eMBErrorCode
eMBRegHoldingCB
                    ( UCHAR *
                                 pucRegBuffer,
                      USHORT     usAddress,
                      USHORT     usNRegs,
                      eMBRegisterMode eMode
                    )
```

Callback function used if a *Holding Register* value is read or written by the protocol stack. The starting register address is given by `usAddress` and the last register is given by `usAddress + usNRegs - 1`.

Parameters:

pucRegBuffer	If the application registers values should be updated the buffer points to the new registers values. If the protocol stack needs to now the current values the callback function should write them into this buffer.
usAddress	The starting address of the register.
usNRegs	Number of registers to read or write.
eMode	If **eMBRegisterMode::MB_REG_WRITE** the application register values should be updated from the values in the buffer. For example this would be the case when the Modbus master has issued an **WRITE SINGLE REGISTER** command. If the value **eMBRegisterMode::MB_REG_READ** the application should copy the current values into the buffer `pucRegBuffer`.

Returns:

The function must return one of the following error codes:

- **eMBErrorCode::MB_ENOERR** If no error occurred. In this case a normal Modbus response is sent.
- **eMBErrorCode::MB_ENOREG** If the application can not supply values for registers within this range. In this case a **ILLEGAL DATA ADDRESS** exception frame is sent as a response.
- **eMBErrorCode::MB_ETIMEDOUT** If the requested register block is currently not available and the application dependent response timeout would be violated. In this case a **SLAVE DEVICE BUSY** exception is sent as a response.
- **eMBErrorCode::MB_EIO** If an unrecoverable error occurred. In this case a **SLAVE DEVICE FAILURE** exception is sent as a response.

Examples:

AVR/demo.c, **LINUX/demo.c**, **MCF5235/demo.c**, **MCF5235TCP/demo.c**, **MSP430/demo.c**, **STR71X/simple2.c**, **STR71XTCP/demo.c**, **WIN32/demo.cpp**, and **WIN32TCP/demo.cpp**.

```
eMBErrorCode
eMBRegInputCB
                ( UCHAR *
                          pucRegBuffer,
                  USHORT usAddress,
                  USHORT usNRegs
                )
```

Callback function used if the value of a *Input Register* is required by the protocol stack. The starting register address is given by `usAddress` and the last register is given by `usAddress + usNRegs - 1`.

Parameters:

pucRegBuffer	A buffer where the callback function should write the current value of the modbus registers to.
usAddress	The starting address of the register. Input registers are in the range 1 - 65535.
usNRegs	Number of registers the callback function must supply.

Returns:

The function must return one of the following error codes:

- **eMBErrorCode::MB_ENOERR** If no error occurred. In this case a normal Modbus response is sent.
- **eMBErrorCode::MB_ENOREG** If the application can not supply values for registers within this range. In this case a **ILLEGAL DATA ADDRESS** exception frame is sent as a response.
- **eMBErrorCode::MB_ETIMEDOUT** If the requested register block is currently not available and the application dependent response timeout would be violated. In this case a **SLAVE DEVICE BUSY** exception is sent as a response.
- **eMBErrorCode::MB_EIO** If an unrecoverable error occurred. In this case a **SLAVE DEVICE FAILURE** exception is sent as a response.

Examples:

AVR/demo.c, **LINUX/demo.c**, **MCF5235/demo.c**, **MCF5235TCP/demo.c**, **MSP430/demo.c**, **STR71X/simple2.c**, **STR71XTCP/demo.c**, **WIN32/demo.cpp**, and **WIN32TCP/demo.cpp**.

Modbus Configuration

Detailed Description

Most modules in the protocol stack are completly optional and can be excluded. This is specially important if target resources are very small and program memory space should be saved.

All of these settings are available in the file **mbconfig.h**

Defines

#define **MB_ASCII_ENABLED** (1)
#define **MB_RTU_ENABLED** (1)
#define **MB_TCP_ENABLED** (0)
#define **MB_ASCII_TIMEOUT_SEC** (1)
#define **MB_FUNC_HANDLERS_MAX** (16)
#define **MB_FUNC_OTHER_REP_SLAVEID_BUF** (32)
#define **MB_FUNC_OTHER_REP_SLAVEID_ENABLED** (1)
#define **MB_FUNC_READ_INPUT_ENABLED** (1)
#define **MB_FUNC_READ_HOLDING_ENABLED** (1)
#define **MB_FUNC_WRITE_HOLDING_ENABLED** (1)
#define **MB_FUNC_WRITE_MULTIPLE_HOLDING_ENABLED** (1)
#define **MB_FUNC_READ_COILS_ENABLED** (1)
#define **MB_FUNC_WRITE_COIL_ENABLED** (1)
#define **MB_FUNC_WRITE_MULTIPLE_COILS_ENABLED** (1)
#define **MB_FUNC_READ_DISCRETE_INPUTS_ENABLED** (1)
#define **MB_FUNC_READWRITE_HOLDING_ENABLED** (1)

Define Documentation

#define
MB_ASCII_ENABLED (1)

If Modbus ASCII support is enabled.

#define
MB_ASCII_TIMEOUT_SEC (1)

The character timeout value for Modbus ASCII.

The character timeout value is not fixed for Modbus ASCII and is therefore a configuration option. It should be set to the maximum expected delay time of the network.

#define
MB_FUNC_HANDLERS_MAX (16)

Maximum number of Modbus functions codes the protocol stack should support.

The maximum number of supported Modbus functions must be greater than the sum of all enabled functions in this file and custom function handlers. If set to small adding more functions will fail.

#define
MB_FUNC_OTHER_REP_SLAVEID_BUF (32)

Number of bytes which should be allocated for the *Report Slave ID* command.

This number limits the maximum size of the additional segment in the report slave id function. See **eMBSetSlaveID( )** for more information on how to set this value. It is only used if MB_FUNC_OTHER_REP_SLAVEID_ENABLED is set to 1.

#define
MB_FUNC_OTHER_REP_SLAVEID_ENABLED (1)

If the *Report Slave ID* function should be enabled.

#define
MB_FUNC_READ_COILS_ENABLED (1)

If the *Read Coils* function should be enabled.

#define
MB_FUNC_READ_DISCRETE_INPUTS_ENABLED (1)

If the *Read Discrete Inputs* function should be enabled.

#define
MB_FUNC_READ_HOLDING_ENABLED (1)

If the *Read Holding Registers* function should be enabled.

#define
MB_FUNC_READ_INPUT_ENABLED (1)

If the *Read Input Registers* function should be enabled.

#define
MB_FUNC_READWRITE_HOLDING_ENABLED (1)

If the *Read/Write Multiple Registers* function should be enabled.

#define
MB_FUNC_WRITE_COIL_ENABLED (1)

If the *Write Coils* function should be enabled.

#define
MB_FUNC_WRITE_HOLDING_ENABLED (1)

If the *Write Single Register* function should be enabled.

#define
MB_FUNC_WRITE_MULTIPLE_COILS_ENABLED (1)

If the *Write Multiple Coils* function should be enabled.

#define
MB_FUNC_WRITE_MULTIPLE_HOLDING_ENABLED (1)

If the *Write Multiple registers* function should be enabled.

#define
MB_RTU_ENABLED (1)

If Modbus RTU support is enabled.

#define
MB_TCP_ENABLED (0)

If Modbus TCP support is enabled.

Automatically generated by Doxygen 1.3.9.1 on 19 Nov 2006.

Utilities

Detailed Description

This module contains some utility functions which can be used by the application. It includes some special functions for working with bitfields backed by a character array buffer.

Functions

void **xMBUtilSetBits** (UCHAR *ucByteBuf, USHORT usBitOffset, UCHAR ucNBits, UCHAR ucValues)

UCHAR **xMBUtilGetBits** (UCHAR *ucByteBuf, USHORT usBitOffset, UCHAR ucNBits)

Function Documentation

UCHAR xMBUtilGetBits (**UCHAR *** *ucByteBuf*, **USHORT** *usBitOffset*, **UCHAR** *ucNBits*)

Function to read bits in a byte buffer.

This function is used to extract up bit values from an array. Up to eight bit values can be extracted in one step.

Parameters:

ucByteBuf	A buffer where the bit values are stored.
usBitOffset	The starting address of the bits to set. The first bit has the offset 0.
ucNBits	Number of bits to modify. The value must always be smaller than 8.

```
UCHAR ucBits[2] = {0, 0};
UCHAR ucResult;

// Extract the bits 3 - 10.
ucResult = xMBUtilGetBits( ucBits, 3, 8 );
```

```
void            ( UCHAR *  ucByteBuf,
xMBUtilSetBits
                  USHORT   usBitOffset,
                  UCHAR    ucNBits,
                  UCHAR    ucValues
                )
```

Function to set bits in a byte buffer.

This function allows the efficient use of an array to implement bitfields. The array used for storing the bits must always be a multiple of two bytes. Up to eight bits can be set or cleared in one operation.

Parameters:

ucByteBuf	A buffer where the bit values are stored. Must be a multiple of 2 bytes. No length checking is performed and if usBitOffset / 8 is greater than the size of the buffer memory contents is overwritten.
usBitOffset	The starting address of the bits to set. The first bit has the offset 0.
ucNBits	Number of bits to modify. The value must always be smaller than 8.
ucValues	Thew new values for the bits. The value for the first bit starting at `usBitOffset` is the LSB of the value `ucValues`

```
ucBits[2] = {0, 0};

// Set bit 4 to 1 (read: set 1 bit starting at bit offset 4 to value 1)
xMBUtilSetBits( ucBits, 4, 1, 1 );

// Set bit 7 to 1 and bit 8 to 0.
xMBUtilSetBits( ucBits, 7, 2, 0x01 );

// Set bits 8 - 11 to 0x05 and bits 12 - 15 to 0x0A;
xMBUtilSetBits( ucBits, 8, 8, 0x5A);
```

Automatically generated by Doxygen 1.3.9.1 on 19 Nov 2006.

FreeModbus Examples

Here is a list of all examples:

- **AVR/demo.c**
- **LINUX/demo.c**
- **MCF5235/demo.c**
- **MCF5235TCP/demo.c**
- **MSP430/demo.c**
- **STR71X/simple2.c**
- **STR71XTCP/demo.c**
- **WIN32/demo.cpp**
- **WIN32TCP/demo.cpp**

Automatically generated by Doxygen 1.3.9.1 on 19 Nov 2006.

AVR/demo.c

```
/*
 * FreeModbus Libary: AVR Demo Application
 * Copyright (C) 2006 Christian Walter <wolti@sil.at>
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA  02110-1301  USA
 *
 * File: $Id: demo.c,v 1.7 2006/06/15 15:38:02 wolti Exp $
 */

/* ----------------------- AVR includes ------------------------------------*/
#include "avr/io.h"
#include "avr/interrupt.h"

/* ----------------------- Modbus includes ----------------------------------*/
#include "mb.h"
#include "mbport.h"

/* ----------------------- Defines ------------------------------------------*/
#define REG_INPUT_START 1000
#define REG_INPUT_NREGS 4

/* ----------------------- Static variables ---------------------------------*/
static USHORT   usRegInputStart = REG_INPUT_START;
static USHORT   usRegInputBuf[REG_INPUT_NREGS];

/* ----------------------- Start implementation -----------------------------*/
int
main( void )
{
    const UCHAR     ucSlaveID[] = { 0xAA, 0xBB, 0xCC };
    eMBErrorCode    eStatus;

    eStatus = eMBInit( MB_RTU, 0x0A, 0, 38400, MB_PAR_EVEN );

    eStatus = eMBSetSlaveID( 0x34, TRUE, ucSlaveID, 3 );
    sei(  );
```

```
    /* Enable the Modbus Protocol Stack. */
    eStatus = eMBEnable(  );

    for( ;; )
    {
        ( void )eMBPoll(  );

        /* Here we simply count the number of poll cycles. */
        usRegInputBuf[0]++;
    }
}

eMBErrorCode
eMBRegInputCB( UCHAR * pucRegBuffer, USHORT usAddress, USHORT usNRegs )
{
    eMBErrorCode    eStatus = MB_ENOERR;
    int             iRegIndex;

    if( ( usAddress >= REG_INPUT_START )
        && ( usAddress + usNRegs <= REG_INPUT_START + REG_INPUT_NREGS ) )
    {
        iRegIndex = ( int )( usAddress - usRegInputStart );
        while( usNRegs > 0 )
        {
            *pucRegBuffer++ =
                ( unsigned char )( usRegInputBuf[iRegIndex] >> 8 );
            *pucRegBuffer++ =
                ( unsigned char )( usRegInputBuf[iRegIndex] & 0xFF );
            iRegIndex++;
            usNRegs--;
        }
    }
    else
    {
        eStatus = MB_ENOREG;
    }

    return eStatus;
}

eMBErrorCode
eMBRegHoldingCB( UCHAR * pucRegBuffer, USHORT usAddress, USHORT usNRegs,
                 eMBRegisterMode eMode )
{
    return MB_ENOREG;
}


eMBErrorCode
eMBRegCoilsCB( UCHAR * pucRegBuffer, USHORT usAddress, USHORT usNCoils,
               eMBRegisterMode eMode )
```

```
{
    return MB_ENOREG;
}

eMBErrorCode
eMBRegDiscreteCB( UCHAR * pucRegBuffer, USHORT usAddress, USHORT usNDiscrete )
{
    return MB_ENOREG;
}
```

Automatically generated by Doxygen 1.3.9.1 on 19 Nov 2006.

LINUX/demo.c

```
/*
 * FreeModbus Libary: Linux Demo Application
 * Copyright (C) 2006 Christian Walter <wolti@sil.at>
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA  02110-1301  USA
 *
 * File: $Id: demo.c,v 1.2 2006/10/12 08:12:06 wolti Exp $
 */

/* ----------------------- Standard includes ----------------------------------*/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <pthread.h>
#include <signal.h>

/* ----------------------- Modbus includes ----------------------------------*/
#include "mb.h"
#include "mbport.h"

/* ----------------------- Defines ------------------------------------------*/
#define PROG            "freemodbus"

#define REG_INPUT_START 1000
#define REG_INPUT_NREGS 4
#define REG_HOLDING_START 2000
#define REG_HOLDING_NREGS 130

/* ----------------------- Static variables ---------------------------------*/
static USHORT   usRegInputStart = REG_INPUT_START;
static USHORT   usRegInputBuf[REG_INPUT_NREGS];
static USHORT   usRegHoldingStart = REG_HOLDING_START;
static USHORT   usRegHoldingBuf[REG_HOLDING_NREGS];
```

```
static enum ThreadState
{
    STOPPED,
    RUNNING,
    SHUTDOWN
} ePollThreadState;

static pthread_mutex_t xLock = PTHREAD_MUTEX_INITIALIZER;
static BOOL     bDoExit;

/* ----------------------- Static functions ---------------------------------*/
static BOOL     bCreatePollingThread( void );
static enum ThreadState eGetPollingThreadState( void );
static void     vSetPollingThreadState( enum ThreadState eNewState );
static void    *pvPollingThread( void *pvParameter );

/* ----------------------- Start implementation -----------------------------*/
BOOL
bSetSignal( int iSignalNr, void ( *pSigHandler ) ( int ) )
{
    BOOL            bResult;
    struct sigaction xNewSig, xOldSig;

    xNewSig.sa_handler = pSigHandler;
    sigemptyset( &xNewSig.sa_mask );
    xNewSig.sa_flags = 0;
    if( sigaction( iSignalNr, &xNewSig, &xOldSig ) != 0 )
    {
        bResult = FALSE;
    }
    else
    {
        bResult = TRUE;
    }
    return bResult;
}

void
vSigShutdown( int xSigNr )
{
    switch ( xSigNr )
    {
    case SIGQUIT:
    case SIGINT:
    case SIGTERM:
        vSetPollingThreadState( SHUTDOWN );
        bDoExit = TRUE;
    }
}

int
main( int argc, char *argv[] )
{
```

```
    int             iExitCode;
    CHAR            cCh;

    const UCHAR     ucSlaveID[] = { 0xAA, 0xBB, 0xCC };
    if( !bSetSignal( SIGQUIT, vSigShutdown ) ||
        !bSetSignal( SIGINT, vSigShutdown ) || !bSetSignal( SIGTERM,
vSigShutdown ) )
    {
        fprintf( stderr, "%s: can't install signal handlers: %s!\n", PROG,
strerror( errno ) );
        iExitCode = EXIT_FAILURE;
    }
    else if( eMBInit( MB_RTU, 0x0A, 0, 38400, MB_PAR_EVEN ) != MB_ENOERR )
    {
        fprintf( stderr, "%s: can't initialize modbus stack!\n", PROG );
        iExitCode = EXIT_FAILURE;
    }
    else if( eMBSetSlaveID( 0x34, TRUE, ucSlaveID, 3 ) != MB_ENOERR )
    {
        fprintf( stderr, "%s: can't set slave id!\n", PROG );
        iExitCode = EXIT_FAILURE;
    }
    else
    {
        vSetPollingThreadState( STOPPED );

        /* CLI interface. */
        printf( "Type 'q' for quit or 'h' for help!\n" );
        bDoExit = FALSE;
        do
        {
            printf( "> " );
            cCh = getchar(  );

            switch ( cCh )
            {
            case 'q':
                bDoExit = TRUE;
                break;
            case 'd':
                vSetPollingThreadState( SHUTDOWN );
                break;
            case 'e':
                if( bCreatePollingThread(  ) != TRUE )
                {
                    printf( "Can't start protocol stack! Already running?\n" );
                }
                break;
            case 's':
                switch ( eGetPollingThreadState(  ) )
                {
                case RUNNING:
                    printf( "Protocol stack is running.\n" );
```

```
                    break;
                case STOPPED:
                    printf( "Protocol stack is stopped.\n" );
                    break;
                case SHUTDOWN:
                    printf( "Protocol stack is shuting down.\n" );
                    break;
                }
                break;
            case 'h':
                printf( "FreeModbus demo application help:\n" );
                printf( "  'd' ... disable protocol stack.\n" );
                printf( "  'e' ... enabled the protocol stack.\n" );
                printf( "  's' ... show current status.\n" );
                printf( "  'q' ... quit application.\n" );
                printf( "  'h' ... this information.\n" );
                printf( "\n" );
                printf( "Copyright 2006 Christian Walter <wolti@sil.at>\n" );
                break;
            default:
                if( !bDoExit && ( cCh != '\n' ) )
                {
                    printf( "illegal command '%c'!\n", cCh );
                }
                break;
            }

            /* eat up everything untill return character. */
            while( !bDoExit && ( cCh != '\n' ) )
            {
                cCh = getchar(  );
            }
        }
        while( !bDoExit );

        /* Release hardware resources. */
        ( void )eMBClose(  );
        iExitCode = EXIT_SUCCESS;
    }
    return iExitCode;
}

BOOL
bCreatePollingThread( void )
{
    BOOL            bResult;
    pthread_t       xThread;

    if( eGetPollingThreadState(  ) == STOPPED )
    {
        if( pthread_create( &xThread, NULL, pvPollingThread, NULL ) != 0 )
        {
            bResult = FALSE;
```

```
        }
        else
        {
            bResult = TRUE;
        }
    }
    else
    {
        bResult = FALSE;
    }

    return bResult;
}

void           *
pvPollingThread( void *pvParameter )
{
    vSetPollingThreadState( RUNNING );

    if( eMBEnable(  ) == MB_ENOERR )
    {
        do
        {
            if( eMBPoll(  ) != MB_ENOERR )
                break;
            usRegInputBuf[0] = ( USHORT ) rand(  );
        }
        while( eGetPollingThreadState(  ) != SHUTDOWN );
    }

    ( void )eMBDisable(  );

    vSetPollingThreadState( STOPPED );

    return 0;
}

enum ThreadState
eGetPollingThreadState(  )
{
    enum ThreadState eCurState;

    ( void )pthread_mutex_lock( &xLock );
    eCurState = ePollThreadState;
    ( void )pthread_mutex_unlock( &xLock );

    return eCurState;
}

void
vSetPollingThreadState( enum ThreadState eNewState )
{
    ( void )pthread_mutex_lock( &xLock );
```

```
    ePollThreadState = eNewState;
    ( void )pthread_mutex_unlock( &xLock );
}

eMBErrorCode
eMBRegInputCB( UCHAR * pucRegBuffer, USHORT usAddress, USHORT usNRegs )
{
    eMBErrorCode    eStatus = MB_ENOERR;
    int             iRegIndex;

    if( ( usAddress >= REG_INPUT_START )
        && ( usAddress + usNRegs <= REG_INPUT_START + REG_INPUT_NREGS ) )
    {
        iRegIndex = ( int )( usAddress - usRegInputStart );
        while( usNRegs > 0 )
        {
            *pucRegBuffer++ = ( unsigned char )( usRegInputBuf[iRegIndex] >>
8 );
            *pucRegBuffer++ = ( unsigned char )( usRegInputBuf[iRegIndex] &
0xFF );
            iRegIndex++;
            usNRegs--;
        }
    }
    else
    {
        eStatus = MB_ENOREG;
    }

    return eStatus;
}

eMBErrorCode
eMBRegHoldingCB( UCHAR * pucRegBuffer, USHORT usAddress, USHORT usNRegs,
eMBRegisterMode eMode )
{
    eMBErrorCode    eStatus = MB_ENOERR;
    int             iRegIndex;

    if( ( usAddress >= REG_HOLDING_START ) &&
        ( usAddress + usNRegs <= REG_HOLDING_START + REG_HOLDING_NREGS ) )
    {
        iRegIndex = ( int )( usAddress - usRegHoldingStart );
        switch ( eMode )
        {
            /* Pass current register values to the protocol stack. */
        case MB_REG_READ:
            while( usNRegs > 0 )
            {
                *pucRegBuffer++ = ( UCHAR ) ( usRegHoldingBuf[iRegIndex] >> 8 );
                *pucRegBuffer++ = ( UCHAR ) ( usRegHoldingBuf[iRegIndex] &
0xFF );
```

```
                iRegIndex++;
                usNRegs--;
            }
            break;

            /* Update current register values with new values from the
             * protocol stack. */
        case MB_REG_WRITE:
            while( usNRegs > 0 )
            {
                usRegHoldingBuf[iRegIndex] = *pucRegBuffer++ << 8;
                usRegHoldingBuf[iRegIndex] |= *pucRegBuffer++;
                iRegIndex++;
                usNRegs--;
            }
        }
    }
    else
    {
        eStatus = MB_ENOREG;
    }
    return eStatus;
}


eMBErrorCode
eMBRegCoilsCB( UCHAR * pucRegBuffer, USHORT usAddress, USHORT usNCoils,
eMBRegisterMode eMode )
{
    return MB_ENOREG;
}

eMBErrorCode
eMBRegDiscreteCB( UCHAR * pucRegBuffer, USHORT usAddress, USHORT usNDiscrete )
{
    return MB_ENOREG;
}
```

Automatically generated by Doxygen 1.3.9.1 on 19 Nov 2006.

MCF5235/demo.c

```
/*
 * FreeModbus Libary: MCF5235 Demo Application
 * Copyright (C) 2006 Christian Walter <wolti@sil.at>
 * Parts of crt0.S Copyright (c) 1995, 1996, 1998 Cygnus Support
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA  02110-1301  USA
 *
 * File: $Id: demo.c,v 1.2 2006/06/15 15:38:55 wolti Exp $
 */

#include "mcf5xxx.h"
#include "mcf523x.h"

#include "mb.h"
#include "mbport.h"

/* ----------------------- Modbus includes ----------------------------------*/
#include "mb.h"
#include "mbport.h"

/* ----------------------- Defines ------------------------------------------*/
#define REG_INPUT_START 1000
#define REG_INPUT_NREGS 4

/* ----------------------- Static variables ---------------------------------*/
static USHORT   usRegInputStart = REG_INPUT_START;
static USHORT   usRegInputBuf[REG_INPUT_NREGS];


int
main( int argc, char *argv[], char *envp[] )
{
    //xMBPortSerialInit (9600UL, 8, MB_PAR_EVEN);
    //vMBPortSerialEnable (TRUE , FALSE);
    //xMBPortTimersInit( 200 );
    //vMBPortTimersEnable();
```

```
    //
    const UCHAR     ucSlaveID[] = { 0xAA, 0xBB, 0xCC };
    eMBErrorCode    eStatus;

    eStatus = eMBInit( MB_RTU, 0x0A, 0, 38400, MB_PAR_EVEN );

    eStatus = eMBSetSlaveID( 0x34, TRUE, ucSlaveID, 3 );

    /* Enable the Modbus Protocol Stack. */
    eStatus = eMBEnable(  );

    for( ;; )
    {
        ( void )eMBPoll(  );

        /* Here we simply count the number of poll cycles. */
        usRegInputBuf[0]++;
    }
    return 0;
}

eMBErrorCode
eMBRegInputCB( UCHAR * pucRegBuffer, USHORT usAddress, USHORT usNRegs )
{
    eMBErrorCode    eStatus = MB_ENOERR;
    int             iRegIndex;

    if( ( usAddress >= REG_INPUT_START )
        && ( usAddress + usNRegs <= REG_INPUT_START + REG_INPUT_NREGS ) )
    {
        iRegIndex = ( int )( usAddress - usRegInputStart );
        while( usNRegs > 0 )
        {
            *pucRegBuffer++ =
                ( unsigned char )( usRegInputBuf[iRegIndex] >> 8 );
            *pucRegBuffer++ =
                ( unsigned char )( usRegInputBuf[iRegIndex] & 0xFF );
            iRegIndex++;
            usNRegs--;
        }
    }
    else
    {
        eStatus = MB_ENOREG;
    }

    return eStatus;
}

eMBErrorCode
eMBRegHoldingCB( UCHAR * pucRegBuffer, USHORT usAddress, USHORT usNRegs,
                 eMBRegisterMode eMode )
```

```
{
    return MB_ENOREG;
}


eMBErrorCode
eMBRegCoilsCB( UCHAR * pucRegBuffer, USHORT usAddress, USHORT usNCoils,
               eMBRegisterMode eMode )
{
    return MB_ENOREG;
}

eMBErrorCode
eMBRegDiscreteCB( UCHAR * pucRegBuffer, USHORT usAddress, USHORT usNDiscrete )
{
    return MB_ENOREG;
}
```

Automatically generated by Doxygen 1.3.9.1 on 19 Nov 2006.

MCF5235TCP/demo.c

```
/* ----------------------- System includes ---------------------------- */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <assert.h>

/* ----------------------- FreeRTOS includes ---------------------------- */
#include "FreeRTOS.h"
#include "task.h"

/* ----------------------- LWIP includes ---------------------------- */
#include "lwip/api.h"
#include "lwip/tcpip.h"
#include "lwip/memp.h"

/* ----------------------- FreeModbus includes ---------------------------- */
#include "mb.h"

/* ----------------------- Project includes ---------------------------- */
#include "mcf5xxx.h"
#include "mcf523x.h"
#include "netif/fec.h"

#include "serial.h"

/* ----------------------- Defines ------------------------------------ */
#define mainCOM_TEST_BAUD_RATE  ( ( unsigned portLONG ) 38400 )

#define mainMB_TASK_PRIORITY    ( tskIDLE_PRIORITY + 3 )
#define PROG                    "FreeModbus"
#define REG_INPUT_START         1000
#define REG_INPUT_NREGS         4
#define REG_HOLDING_START       2000
#define REG_HOLDING_NREGS       130

/* ----------------------- Static variables ---------------------------------*/
static USHORT   usRegInputStart = REG_INPUT_START;
static USHORT   usRegInputBuf[REG_INPUT_NREGS];
static USHORT   usRegHoldingStart = REG_HOLDING_START;
static USHORT   usRegHoldingBuf[REG_HOLDING_NREGS];

xComPortHandle  xSTDComPort = NULL;

/* ----------------------- Static functions ---------------------------- */
static void     vlwIPInit( void );
static void     vMBServerTask( void *arg );

/* ----------------------- Implementation ---------------------------- */
```

```
int
main( int argc, char *argv[] )
{
    asm volatile    ( "move.w  #0x2000, %sr\n\t" );

    /* Initialize serial communication device. */
    xSTDComPort = xSerialPortInitMinimal( 38400, 8 );

    /* Initialize lwIP protocol stack. */
    vlwIPInit(  );

    if( sys_thread_new( vMBServerTask, NULL, mainMB_TASK_PRIORITY ) == NULL )
    {
        fprintf( stderr, "%s: can't create modbus task!\r\n", PROG );
    }
    else
    {
        /* Now all the tasks have been started - start the scheduler. */
        vTaskStartScheduler(  );
    }

    /* Should never get here! */
    return 0;
}

void
vlwIPInit( void )
{
    /* Initialize lwIP and its interface layer. */
    sys_init(  );
    mem_init(  );
    memp_init(  );
    pbuf_init(  );
    netif_init(  );
    ip_init(  );
    tcpip_init( NULL, NULL );
}

void
vMBServerTask( void *arg )
{
    eMBErrorCode    xStatus;
    struct ip_addr  xIpAddr, xNetMast, xGateway;
    struct netif    xFEC523x;

    IP4_ADDR( &xIpAddr, 10, 0, 10, 2 );
    IP4_ADDR( &xNetMast, 255, 255, 255, 0 );
    IP4_ADDR( &xGateway, 10, 0, 10, 1 );
    netif_add( &xFEC523x, &xIpAddr, &xNetMast, &xGateway, NULL,
mcf523xfec_init, tcpip_input );
    /* Make it the default interface */
    netif_set_default( &xFEC523x );
    /* Bring it up */
```

```
    netif_set_up( &xFEC523x );


    for( ;; )
    {
        if( eMBTCPInit( MB_TCP_PORT_USE_DEFAULT ) != MB_ENOERR )
        {
            fprintf( stderr, "%s: can't initialize modbus stack!\r\n", PROG );
        }
        else if( eMBEnable(  ) != MB_ENOERR )
        {
            fprintf( stderr, "%s: can't enable modbus stack!\r\n", PROG );
        }
        else
        {
            do
            {
                xStatus = eMBPoll(  );
            }
            while( xStatus == MB_ENOERR );
        }
        /* An error occured. Maybe we can restart. */
        ( void )eMBDisable(  );
        ( void )eMBClose(  );
    }
}

eMBErrorCode
eMBRegInputCB( UCHAR * pucRegBuffer, USHORT usAddress, USHORT usNRegs )
{
    eMBErrorCode    eStatus = MB_ENOERR;
    int             iRegIndex;

    if( ( usAddress >= REG_INPUT_START )
        && ( usAddress + usNRegs <= REG_INPUT_START + REG_INPUT_NREGS ) )
    {
        iRegIndex = ( int )( usAddress - usRegInputStart );
        while( usNRegs > 0 )
        {
            *pucRegBuffer++ = ( unsigned char )( usRegInputBuf[iRegIndex] >>
8 );
            *pucRegBuffer++ = ( unsigned char )( usRegInputBuf[iRegIndex] &
0xFF );
            iRegIndex++;
            usNRegs--;
        }
    }
    else
    {
        eStatus = MB_ENOREG;
    }
    return eStatus;
```

```
}

eMBErrorCode
eMBRegHoldingCB( UCHAR * pucRegBuffer, USHORT usAddress, USHORT usNRegs,
eMBRegisterMode eMode )
{
    eMBErrorCode    eStatus = MB_ENOERR;
    int             iRegIndex;

    if( ( usAddress >= REG_HOLDING_START ) &&
        ( usAddress + usNRegs <= REG_HOLDING_START + REG_HOLDING_NREGS ) )
    {
        iRegIndex = ( int )( usAddress - usRegHoldingStart );
        switch ( eMode )
        {
            /* Pass current register values to the protocol stack. */
        case MB_REG_READ:
            while( usNRegs > 0 )
            {
                *pucRegBuffer++ = ( UCHAR ) ( usRegHoldingBuf[iRegIndex] >> 8 );
                *pucRegBuffer++ = ( UCHAR ) ( usRegHoldingBuf[iRegIndex] &
0xFF );
                iRegIndex++;
                usNRegs--;
            }
            break;

            /* Update current register values with new values from the
             * protocol stack. */
        case MB_REG_WRITE:
            while( usNRegs > 0 )
            {
                usRegHoldingBuf[iRegIndex] = *pucRegBuffer++ << 8;
                usRegHoldingBuf[iRegIndex] |= *pucRegBuffer++;
                iRegIndex++;
                usNRegs--;
            }
        }
    }
    else
    {
        eStatus = MB_ENOREG;
    }
    return eStatus;
}

eMBErrorCode
eMBRegCoilsCB( UCHAR * pucRegBuffer, USHORT usAddress, USHORT usNCoils,
eMBRegisterMode eMode )
{
    return MB_ENOREG;
}
```

```
eMBErrorCode
eMBRegDiscreteCB( UCHAR * pucRegBuffer, USHORT usAddress, USHORT usNDiscrete )
{
    return MB_ENOREG;
}
```

MSP430/demo.c

```
/*
 * FreeModbus Libary: MSP430 Demo Application
 * Copyright (C) 2006 Christian Walter <wolti@sil.at>
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA  02110-1301  USA
 *
 * File: $Id: demo.c,v 1.2 2006/11/19 03:36:01 wolti Exp $
 */

/* ----------------------- Platform includes --------------------------------*/
#include "port.h"
#include "dco.h"

/* ----------------------- Modbus includes ----------------------------------*/
#include "mb.h"
#include "mbport.h"

/* ----------------------- Defines ------------------------------------------*/
#define REG_INPUT_START   1000
#define REG_INPUT_NREGS   4
#define REG_HOLDING_START 1000
#define REG_HOLDING_NREGS 130

/* ----------------------- Static variables ---------------------------------*/
static USHORT   usRegInputStart = REG_INPUT_START;
static USHORT   usRegInputBuf[REG_INPUT_NREGS];
static USHORT   usRegHoldingStart = REG_HOLDING_START;
static USHORT   usRegHoldingBuf[REG_HOLDING_NREGS];

/* ----------------------- Start implementation -----------------------------*/
int
main( void )
{
    eMBErrorCode    eStatus;
    volatile USHORT usACLKCnt;
```

```
    /* Stop Watchdog Timer. */
    WDTCTL = WDTPW + WDTHOLD;

    /* Delay for ACLK startup. */
    for( usACLKCnt = 0xFFFF; usACLKCnt != 0; usACLKCnt-- );
    if( cTISetDCO( TI_DCO_4MHZ ) == TI_DCO_NO_ERROR )
    {
        _EINT(  );

        /* Initialize Protocol Stack. */
        if( ( eStatus = eMBInit( MB_ASCII, 0x0A, 0, 38400, MB_PAR_ODD ) ) !=
MB_ENOERR )
        {
        }
        /* Enable the Modbus Protocol Stack. */
        else if( ( eStatus = eMBEnable(  ) ) != MB_ENOERR )
        {
        }
        else
        {
            for( ;; )
            {
                ( void )eMBPoll(  );

                /* Here we simply count the number of poll cycles. */
                usRegInputBuf[0]++;
            }
        }
    }
    for( ;; );
}

eMBErrorCode
eMBRegInputCB( UCHAR * pucRegBuffer, USHORT usAddress, USHORT usNRegs )
{
    eMBErrorCode    eStatus = MB_ENOERR;
    int             iRegIndex;

    if( ( usAddress >= REG_INPUT_START )
        && ( usAddress + usNRegs <= REG_INPUT_START + REG_INPUT_NREGS ) )
    {
        iRegIndex = ( int )( usAddress - usRegInputStart );
        while( usNRegs > 0 )
        {
            *pucRegBuffer++ = ( unsigned char )( usRegInputBuf[iRegIndex] >>
8 );
            *pucRegBuffer++ = ( unsigned char )( usRegInputBuf[iRegIndex] &
0xFF );
            iRegIndex++;
            usNRegs--;
        }
    }
```

```
    else
    {
        eStatus = MB_ENOREG;
    }

    return eStatus;
}

eMBErrorCode
eMBRegHoldingCB( UCHAR * pucRegBuffer, USHORT usAddress, USHORT usNRegs,
eMBRegisterMode eMode )
{
    eMBErrorCode    eStatus = MB_ENOERR;
    int             iRegIndex;

    if( ( usAddress >= REG_HOLDING_START ) &&
        ( usAddress + usNRegs <= REG_HOLDING_START + REG_HOLDING_NREGS ) )
    {
        iRegIndex = ( int )( usAddress - usRegHoldingStart );
        switch ( eMode )
        {
            /* Pass current register values to the protocol stack. */
        case MB_REG_READ:
            while( usNRegs > 0 )
            {
                *pucRegBuffer++ = ( unsigned char )( usRegHoldingBuf[iRegIndex]
>> 8 );
                *pucRegBuffer++ = ( unsigned char )( usRegHoldingBuf[iRegIndex]
& 0xFF );
                iRegIndex++;
                usNRegs--;
            }
            break;

            /* Update current register values with new values from the
             * protocol stack. */
        case MB_REG_WRITE:
            while( usNRegs > 0 )
            {
                usRegHoldingBuf[iRegIndex] = *pucRegBuffer++ << 8;
                usRegHoldingBuf[iRegIndex] |= *pucRegBuffer++;
                iRegIndex++;
                usNRegs--;
            }
        }
    }
    else
    {
        eStatus = MB_ENOREG;
    }
    return eStatus;
}
```

```
eMBErrorCode
eMBRegCoilsCB( UCHAR * pucRegBuffer, USHORT usAddress, USHORT usNCoils,
eMBRegisterMode eMode )
{
    return MB_ENOREG;
}

eMBErrorCode
eMBRegDiscreteCB( UCHAR * pucRegBuffer, USHORT usAddress, USHORT usNDiscrete )
{
    return MB_ENOREG;
}
```

Automatically generated by Doxygen 1.3.9.1 on 19 Nov 2006.

STR71X/simple2.c

```
/*
 * FreeModbus Libary: STR71x Demo Application
 * Copyright (C) 2006 Christian Walter <wolti@sil.at>
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA  02110-1301  USA
 *
 * File: $Id: simple2.c,v 1.8 2006/05/14 21:54:16 wolti Exp $
 */

/* ----------------------- System includes ----------------------------------*/
#include "assert.h"

/* ----------------------- Platform includes --------------------------------*/
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"

/* ----------------------- STR71X includes ----------------------------------*/
#include "eic.h"

/* ----------------------- Modbus includes ----------------------------------*/
#include "mb.h"

/* ----------------------- Defines ------------------------------------------*/
#define REG_INPUT_START 1000
#define REG_INPUT_NREGS 4

/* ----------------------- Static variables ---------------------------------*/
static unsigned short usRegInputStart = REG_INPUT_START;
static unsigned short usRegInputBuf[REG_INPUT_NREGS];

/* ----------------------- Static functions ---------------------------------*/
static void     vModbusTask( void *pvParameters );

/* ----------------------- Start implementation -----------------------------*/
int
```

```
main( void )
{
    EIC_Init(  );
    EIC_IRQConfig( ENABLE );

    ( void )xTaskCreate( vModbusTask, NULL, configMINIMAL_STACK_SIZE, NULL,
                         tskIDLE_PRIORITY, NULL );

    vTaskStartScheduler(  );
    return 0;
}

static void
vModbusTask( void *pvParameters )
{
    portTickType    xLastWakeTime;

    /* Select either ASCII or RTU Mode. */
    ( void )eMBInit( MB_RTU, 0x0A, 38400, MB_PAR_EVEN );

    /* Enable the Modbus Protocol Stack. */
    ( void )eMBEnable(  );
    for( ;; )
    {
        /* Call the main polling loop of the Modbus protocol stack. */
        ( void )eMBPoll(  );
        /* Application specific actions. Count the number of poll cycles. */
        usRegInputBuf[0]++;
        /* Hold the current FreeRTOS ticks. */
        xLastWakeTime = xTaskGetTickCount(  );
        usRegInputBuf[1] = ( unsigned portSHORT )( xLastWakeTime >> 16UL );
        usRegInputBuf[2] = ( unsigned portSHORT )( xLastWakeTime & 0xFFFFUL );
        /* The constant value. */
        usRegInputBuf[3] = 33;
    }
}

eMBErrorCode
eMBRegInputCB( UCHAR * pucRegBuffer, USHORT usAddress, USHORT usNRegs )
{
    eMBErrorCode    eStatus = MB_ENOERR;
    int             iRegIndex;

    if( ( usAddress >= REG_INPUT_START )
        && ( usAddress + usNRegs <= REG_INPUT_START + REG_INPUT_NREGS ) )
    {
        iRegIndex = ( int )( usAddress - usRegInputStart );
        while( usNRegs > 0 )
        {
            *pucRegBuffer++ =
                ( unsigned char )( usRegInputBuf[iRegIndex] >> 8 );
            *pucRegBuffer++ =
```

```
                ( unsigned char )( usRegInputBuf[iRegIndex] & 0xFF );
            iRegIndex++;
            usNRegs--;
        }
    }
    else
    {
        eStatus = MB_ENOREG;
    }

    return eStatus;
}

eMBErrorCode
eMBRegHoldingCB( UCHAR * pucRegBuffer, USHORT usAddress, USHORT usNRegs,
                 eMBRegisterMode eMode )
{
    return MB_ENOREG;
}


eMBErrorCode
eMBRegCoilsCB( UCHAR * pucRegBuffer, USHORT usAddress, USHORT usNCoils,
               eMBRegisterMode eMode )
{
    return MB_ENOREG;
}

eMBErrorCode
eMBRegDiscreteCB( UCHAR * pucRegBuffer, USHORT usAddress, USHORT usNDiscrete )
{
    return MB_ENOREG;
}

void
__assert( const char *pcFile, const char *pcLine, int iLineNumber )
{
    portENTER_CRITICAL(  );
    for( ;; );
}
```

STR71XTCP/demo.c

```
/*
 * FreeModbus Libary: STR71XTCP Demo Application
 * Copyright (C) 2006 Christian Walter <wolti@sil.at>
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA  02110-1301  USA
 *
 * File: $Id: demo.c,v 1.4 2006/09/13 21:19:46 wolti Exp $
 */

/* ----------------------- System includes ----------------------------------*/
#include <stdio.h>

/* ----------------------- lwIP includes ------------------------------------*/
#include "lwip/opt.h"
#include "lwip/sio.h"
#include "lwip/sys.h"
#include "lwip/inet.h"
#include "ppp/ppp.h"
#include "arch/cc.h"

/* ----------------------- FreeRTOS includes --------------------------------*/
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"

/* ----------------------- Platform includes --------------------------------*/
#include "eic.h"
#include "netif/serial.h"

/* ----------------------- Modbus includes ----------------------------------*/
#include "mb.h"

/* ----------------------- Defines ------------------------------------------*/
#define mainMB_TASK_PRIORITY    ( tskIDLE_PRIORITY + 3 )
#define REG_INPUT_START         1000
#define REG_INPUT_NREGS         4
```

```
#define REG_HOLDING_START       2000
#define REG_HOLDING_NREGS       130

#define PPP_AUTH_ENABLED        1
#define PPP_USER                "freemodbus"
#define PPP_PASS                "insecure"

/* ----------------------- Type definitions ---------------------------------*/
typedef enum
{
    CONNECTING, CONNECTED, DISCONNECT
} ePPPThreadControl;

/* ----------------------- Static variables ---------------------------------*/
static USHORT   usRegInputStart = REG_INPUT_START;
static USHORT   usRegInputBuf[REG_INPUT_NREGS];
static USHORT   usRegHoldingStart = REG_HOLDING_START;
static USHORT   usRegHoldingBuf[REG_HOLDING_NREGS];
static ePPPThreadControl ePPPThrCtl;


/* ----------------------- Static functions ---------------------------------*/
static void     vlwIPInit( void );
static void     vMBServerTask( void *arg );
static void     vPPPStatusCB( void *ctx, int errCode, void *arg );

sio_fd_t        stdio_fd;
sio_fd_t        ppp_fd;

/* ----------------------- Start implementation -----------------------------*/
int
main( void )
{
    EIC_Init(  );
    EIC_IRQConfig( ENABLE );

    /* Use UART0 as stdin/stdout for debug purposes. */
    if( ( stdio_fd = sio_open_new( 0, 115200, 8, SIO_STOP_1, SIO_PAR_NONE ) )
== SIO_FD_NULL )
    {
        /* nothing we can do here - no stdout means no logging. */
    }
    else
    {
        /* Initialize lwIP and its interface layer. */
        vlwIPInit(  );

        /* Use UART1 as PPP device. */
        if( ( ppp_fd = sio_open_new( 1, 115200, 8, SIO_STOP_1, SIO_PAR_NONE ) )
== SIO_FD_NULL )
        {
            vMBPortLog( MB_LOG_ERROR, "PPP", "can't open PPP device!\r\n" );
        }
```

```
        else if( sys_thread_new( vMBServerTask, NULL, mainMB_TASK_PRIORITY ) ==
SYS_THREAD_NULL )
        {
            vMBPortLog( MB_LOG_ERROR, "MB-INIT", "can't start modbus task!\r
\n" );
        }
        else
        {
            vMBPortLog( MB_LOG_INFO, "MB-INIT", "FreeModbus demo application
starting...\r\n" );
            /* Everything ready. Start the scheduler. */
            vTaskStartScheduler(  );
        }
    }

    for( ;; );
}

void
vlwIPInit( void )
{
    sys_init(  );
    mem_init(  );
    memp_init(  );
    pbuf_init(  );
    netif_init(  );
    ip_init(  );
    tcpip_init( NULL, NULL );
}

void
vMBServerTask( void *arg )
{
    eMBErrorCode    xStatus;
    ePPPThreadControl ePPPThrCtlCur;
    int             ppp_con_fd;
    portTickType    xTicks;

    pppInit(  );
    if( PPP_AUTH_ENABLED )
    {
      pppSetAuth( PPPAUTHTYPE_PAP, PPP_USER, PPP_PASS );
    }
    else
    {
      pppSetAuth( PPPAUTHTYPE_NONE, NULL, NULL );
    }
    do
    {
        vPortEnterCritical(  );
        ePPPThrCtl = CONNECTING;
        vPortExitCritical(  );
        if( ( ppp_con_fd = pppOpen( ppp_fd, vPPPStatusCB, NULL ) ) ==
```

```
PPPERR_NONE )
        {
            /* Check every 50ms if the state of the connecton has changed.
             * This could either mean it was aborted or successful.
             */
            do
            {
                vTaskDelay( ( portTickType ) ( 50UL / portTICK_RATE_MS ) );
                vPortEnterCritical(  );
                ePPPThrCtlCur = ePPPThrCtl;
                vPortExitCritical(  );
            }
            while( ePPPThrCtlCur == CONNECTING );

            if( ePPPThrCtlCur == CONNECTED )
            {
                if( eMBTCPInit( MB_TCP_PORT_USE_DEFAULT ) != MB_ENOERR )
                {
                    vMBPortLog( MB_LOG_ERROR, "PPP", "can't initalize modbus
stack!\r\n" );
                }
                else if( eMBEnable(  ) != MB_ENOERR )
                {
                    vMBPortLog( MB_LOG_ERROR, "PPP", "can't enable modbus stack!
\r\n" );
                }
                else
                {
                    do
                    {
                        vPortEnterCritical(  );
                        ePPPThrCtlCur = ePPPThrCtl;
                        vPortExitCritical(  );

                        /* Application code here. */
                        xStatus = eMBPoll(  );

                        /* Update input registers with the current system
tick. */
                        xTicks = xTaskGetTickCount(  );

                        /* Note: little endian stuff */
                        usRegInputBuf[0] = ( USHORT ) ( xTicks );
                        usRegInputBuf[1] = ( USHORT ) ( xTicks >> 16UL );

                    }
                    while( ( xStatus == MB_ENOERR ) && ( ePPPThrCtlCur ==
CONNECTED ) );

                    ( void )eMBDisable(  );
                    ( void )eMBClose(  );
                }
```

```
            }
            /* FIXME: pppClose bugs because thread is not stopped. */
            /* Connection has been closed. */
            pppClose( ppp_con_fd );
        }

        /* Wait 1s until reopening the connection. */
        vTaskDelay( ( portTickType ) ( 1000UL / portTICK_RATE_MS ) );
    }
    while( pdTRUE );
}

void
vPPPStatusCB( void *ctx, int err, void *arg )
{
    /* Imported from ipcp.c */
    extern char    *_inet_ntoa( u32_t n );

    ePPPThreadControl ePPPThrCtlNew;
    struct ppp_addrs *ppp_addrs;

    switch ( err )
    {
        /* No error. */
    case PPPERR_NONE:
        ePPPThrCtlNew = CONNECTED;
        ppp_addrs = arg;
        vMBPortLog( MB_LOG_INFO, "PPP", "new PPP connection established\r\n" );
        vMBPortLog( MB_LOG_INFO, "PPP", "  our IP address = %s\r\n",
                    _inet_ntoa( ppp_addrs->our_ipaddr.addr ) );
        vMBPortLog( MB_LOG_INFO, "PPP", "  his IP address = %s\r\n",
                    _inet_ntoa( ppp_addrs->his_ipaddr.addr ) );
        vMBPortLog( MB_LOG_INFO, "PPP", "  netmask = %s\r\n",
                    _inet_ntoa( ppp_addrs->netmask.addr ) );
        break;
    default:
        ePPPThrCtlNew = DISCONNECT;
        vMBPortLog( MB_LOG_ERROR, "PPP", "PPP connection died ( err = %d )\r
\n", err );
        break;
    }
    vPortEnterCritical(  );
    ePPPThrCtl = ePPPThrCtlNew;
    vPortExitCritical(  );
}

eMBErrorCode
eMBRegInputCB( UCHAR * pucRegBuffer, USHORT usAddress, USHORT usNRegs )
{
    eMBErrorCode    eStatus = MB_ENOERR;
    int             iRegIndex;

    if( ( usAddress >= REG_INPUT_START )
```

```
        && ( usAddress + usNRegs <= REG_INPUT_START + REG_INPUT_NREGS ) )
    {
        iRegIndex = ( int )( usAddress - usRegInputStart );
        while( usNRegs > 0 )
        {
            *pucRegBuffer++ = ( unsigned char )( usRegInputBuf[iRegIndex] >>
8 );
            *pucRegBuffer++ = ( unsigned char )( usRegInputBuf[iRegIndex] &
0xFF );
            iRegIndex++;
            usNRegs--;
        }
    }
    else
    {
        eStatus = MB_ENOREG;
    }
    return eStatus;
}

eMBErrorCode
eMBRegHoldingCB( UCHAR * pucRegBuffer, USHORT usAddress, USHORT usNRegs,
eMBRegisterMode eMode )
{
    eMBErrorCode    eStatus = MB_ENOERR;
    int             iRegIndex;

    if( ( usAddress >= REG_HOLDING_START ) &&
        ( usAddress + usNRegs <= REG_HOLDING_START + REG_HOLDING_NREGS ) )
    {
        iRegIndex = ( int )( usAddress - usRegHoldingStart );
        switch ( eMode )
        {
            /* Pass current register values to the protocol stack. */
        case MB_REG_READ:
            while( usNRegs > 0 )
            {
                *pucRegBuffer++ = ( UCHAR ) ( usRegHoldingBuf[iRegIndex] >> 8 );
                *pucRegBuffer++ = ( UCHAR ) ( usRegHoldingBuf[iRegIndex] &
0xFF );
                iRegIndex++;
                usNRegs--;
            }
            break;

            /* Update current register values with new values from the
             * protocol stack. */
        case MB_REG_WRITE:
            while( usNRegs > 0 )
            {
                usRegHoldingBuf[iRegIndex] = *pucRegBuffer++ << 8;
                usRegHoldingBuf[iRegIndex] |= *pucRegBuffer++;
```

```
                iRegIndex++;
                usNRegs--;
            }
        }
    }
    else
    {
        eStatus = MB_ENOREG;
    }
    return eStatus;
}


eMBErrorCode
eMBRegCoilsCB( UCHAR * pucRegBuffer, USHORT usAddress, USHORT usNCoils,
eMBRegisterMode eMode )
{
    return MB_ENOREG;
}

eMBErrorCode
eMBRegDiscreteCB( UCHAR * pucRegBuffer, USHORT usAddress, USHORT usNDiscrete )
{
    return MB_ENOREG;
}
```

Automatically generated by Doxygen 1.3.9.1 on 19 Nov 2006.

WIN32/demo.cpp

```
/*
 * FreeModbus Libary: Win32 Demo Application
 * Copyright (C) 2006 Christian Walter <wolti@sil.at>
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA  02110-1301  USA
 *
 * File: $Id: demo.cpp,v 1.3 2006/06/26 19:23:40 wolti Exp $
 */

#include "stdafx.h"

/* ----------------------- Modbus includes ----------------------------------*/
#include "mb.h"
#include "mbport.h"

/* ----------------------- Defines ------------------------------------------*/
#define PROG            _T("freemodbus")

#define REG_INPUT_START 1000
#define REG_INPUT_NREGS 4
#define REG_HOLDING_START 2000
#define REG_HOLDING_NREGS 130

/* ----------------------- Static variables ---------------------------------*/
static USHORT   usRegInputStart = REG_INPUT_START;
static USHORT   usRegInputBuf[REG_INPUT_NREGS];
static USHORT   usRegHoldingStart = REG_HOLDING_START;
static USHORT   usRegHoldingBuf[REG_HOLDING_NREGS];

static HANDLE   hPollThread;
static CRITICAL_SECTION hPollLock;
static enum ThreadState
{
    STOPPED,
    RUNNING,
    SHUTDOWN
```

```
} ePollThreadState;

/* ----------------------- Static functions ---------------------------------*/
static BOOL     bCreatePollingThread( void );
static enum ThreadState eGetPollingThreadState( void );
static void     eSetPollingThreadState( enum ThreadState eNewState );
static DWORD WINAPI dwPollingThread( LPVOID lpParameter );

/* ----------------------- Start implementation -----------------------------*/
int
_tmain( int argc, _TCHAR * argv[] )
{
    int             iExitCode;
    TCHAR           cCh;
    BOOL            bDoExit;

    const UCHAR     ucSlaveID[] = { 0xAA, 0xBB, 0xCC };

    if( eMBInit( MB_RTU, 0x0A, 1, 38400, MB_PAR_EVEN ) != MB_ENOERR )
    {
        _ftprintf( stderr, _T( "%s: can't initialize modbus stack!\r\n" ),
PROG );
        iExitCode = EXIT_FAILURE;
    }
    else if( eMBSetSlaveID( 0x34, TRUE, ucSlaveID, 3 ) != MB_ENOERR )
    {
        _ftprintf( stderr, _T( "%s: can't set slave id!\r\n" ), PROG );
        iExitCode = EXIT_FAILURE;
    }
    else
    {
        /* Create synchronization primitives and set the current state
         * of the thread to STOPPED.
         */
        InitializeCriticalSection( &hPollLock );
        eSetPollingThreadState( STOPPED );

        /* CLI interface. */
        _tprintf( _T( "Type 'q' for quit or 'h' for help!\r\n" ) );
        bDoExit = FALSE;
        do
        {
            _tprintf( _T( "> " ) );
            cCh = _gettchar(  );
            switch ( cCh )
            {
            case _TCHAR( 'q' ):
                bDoExit = TRUE;
                break;
            case _TCHAR( 'd' ):
                eSetPollingThreadState( SHUTDOWN );
                break;
            case _TCHAR( 'e' ):
```

```
                if( bCreatePollingThread(  ) != TRUE )
                {
                    _tprintf( _T( "Can't start protocol stack! Already running?
\r\n" ) );
                }
                break;
            case _TCHAR( 's' ):
                switch ( eGetPollingThreadState(  ) )
                {
                case RUNNING:
                    _tprintf( _T( "Protocol stack is running.\r\n" ) );
                    break;
                case STOPPED:
                    _tprintf( _T( "Protocol stack is stopped.\r\n" ) );
                    break;
                case SHUTDOWN:
                    _tprintf( _T( "Protocol stack is shuting down.\r\n" ) );
                    break;
                }
                break;
            case _TCHAR( 'h' ):
                _tprintf( _T( "FreeModbus demo application help:\r\n" ) );
                _tprintf( _T( "  'd' ... disable protocol stack.\r\n" ) );
                _tprintf( _T( "  'e' ... enabled the protocol stack\r\n" ) );
                _tprintf( _T( "  's' ... show current status\r\n" ) );
                _tprintf( _T( "  'q' ... quit applicationr\r\n" ) );
                _tprintf( _T( "  'h' ... this information\r\n" ) );
                _tprintf( _T( "\r\n" ) );
                _tprintf( _T( "Copyright 2006 Christian Walter <wolti@sil.at>\r
\n" ) );
                break;
            default:
                if( cCh != _TCHAR('\n') )
                {
                    _tprintf( _T( "illegal command '%c'!\r\n" ), cCh );
                }
                break;
            }

            /* eat up everything untill return character. */
            while( cCh != '\n' )
            {
                cCh = _gettchar(  );
            }
        }
        while( !bDoExit );

        /* Release hardware resources. */
        ( void )eMBClose(  );
        iExitCode = EXIT_SUCCESS;
    }
    return iExitCode;
}
```

```
BOOL
bCreatePollingThread( void )
{
    BOOL            bResult;

    if( eGetPollingThreadState(  ) == STOPPED )
    {
        if( ( hPollThread = CreateThread( NULL, 0, dwPollingThread, NULL, 0,
NULL ) ) == NULL )
        {
            /* Can't create the polling thread. */
            bResult = FALSE;
        }
        else
        {
            bResult = TRUE;
        }
    }
    else
    {
        bResult = FALSE;
    }

    return bResult;
}

DWORD           WINAPI
dwPollingThread( LPVOID lpParameter )
{
    eSetPollingThreadState( RUNNING );

    if( eMBEnable(  ) == MB_ENOERR )
    {
        do
        {
            if( eMBPoll(  ) != MB_ENOERR )
                break;
        }
        while( eGetPollingThreadState(  ) != SHUTDOWN );
    }

    ( void )eMBDisable(  );

    eSetPollingThreadState( STOPPED );

    return 0;
}

enum ThreadState
eGetPollingThreadState(  )
{
    enum ThreadState eCurState;
```

```
    EnterCriticalSection( &hPollLock );
    eCurState = ePollThreadState;
    LeaveCriticalSection( &hPollLock );

    return eCurState;
}

void
eSetPollingThreadState( enum ThreadState eNewState )
{
    EnterCriticalSection( &hPollLock );
    ePollThreadState = eNewState;
    LeaveCriticalSection( &hPollLock );
}

eMBErrorCode
eMBRegInputCB( UCHAR * pucRegBuffer, USHORT usAddress, USHORT usNRegs )
{
    eMBErrorCode    eStatus = MB_ENOERR;
    int             iRegIndex;

    if( ( usAddress >= REG_INPUT_START )
        && ( usAddress + usNRegs <= REG_INPUT_START + REG_INPUT_NREGS ) )
    {
        iRegIndex = ( int )( usAddress - usRegInputStart );
        while( usNRegs > 0 )
        {
            *pucRegBuffer++ = ( unsigned char )( usRegInputBuf[iRegIndex] >>
8 );
            *pucRegBuffer++ = ( unsigned char )( usRegInputBuf[iRegIndex] &
0xFF );
            iRegIndex++;
            usNRegs--;
        }
    }
    else
    {
        eStatus = MB_ENOREG;
    }

    return eStatus;
}

eMBErrorCode
eMBRegHoldingCB( UCHAR * pucRegBuffer, USHORT usAddress, USHORT usNRegs,
eMBRegisterMode eMode )
{
    eMBErrorCode    eStatus = MB_ENOERR;
    int             iRegIndex;

    if( ( usAddress >= REG_HOLDING_START ) &&
```

```
        ( usAddress + usNRegs <= REG_HOLDING_START + REG_HOLDING_NREGS ) )
    {
        iRegIndex = ( int )( usAddress - usRegHoldingStart );
        switch ( eMode )
        {
            /* Pass current register values to the protocol stack. */
        case MB_REG_READ:
            while( usNRegs > 0 )
            {
                *pucRegBuffer++ = ( UCHAR ) ( usRegHoldingBuf[iRegIndex] >> 8 );
                *pucRegBuffer++ = ( UCHAR ) ( usRegHoldingBuf[iRegIndex] &
0xFF );
                iRegIndex++;
                usNRegs--;
            }
            break;

            /* Update current register values with new values from the
             * protocol stack. */
        case MB_REG_WRITE:
            while( usNRegs > 0 )
            {
                usRegHoldingBuf[iRegIndex] = *pucRegBuffer++ << 8;
                usRegHoldingBuf[iRegIndex] |= *pucRegBuffer++;
                iRegIndex++;
                usNRegs--;
            }
        }
    }
    else
    {
        eStatus = MB_ENOREG;
    }
    return eStatus;
}


eMBErrorCode
eMBRegCoilsCB( UCHAR * pucRegBuffer, USHORT usAddress, USHORT usNCoils,
eMBRegisterMode eMode )
{
    return MB_ENOREG;
}

eMBErrorCode
eMBRegDiscreteCB( UCHAR * pucRegBuffer, USHORT usAddress, USHORT usNDiscrete )
{
    return MB_ENOREG;
}
```

WIN32TCP/demo.cpp

```
/*
 * FreeModbus Libary: Win32 Demo Application
 * Copyright (C) 2006 Christian Walter <wolti@sil.at>
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA  02110-1301  USA
 *
 * File: $Id: demo.cpp,v 1.2 2006/06/26 19:24:07 wolti Exp $
 */

#include "stdafx.h"

/* ----------------------- Modbus includes ----------------------------------*/
#include "mb.h"
#include "mbport.h"

/* ----------------------- Defines ------------------------------------------*/
#define PROG            _T("freemodbus")

#define REG_INPUT_START 1000
#define REG_INPUT_NREGS 4
#define REG_HOLDING_START 2000
#define REG_HOLDING_NREGS 130

/* ----------------------- Static variables ---------------------------------*/
static USHORT   usRegInputStart = REG_INPUT_START;
static USHORT   usRegInputBuf[REG_INPUT_NREGS];
static USHORT   usRegHoldingStart = REG_HOLDING_START;
static USHORT   usRegHoldingBuf[REG_HOLDING_NREGS];

static HANDLE   hPollThread;
static CRITICAL_SECTION hPollLock;
static enum ThreadState
{
    STOPPED,
    RUNNING,
    SHUTDOWN
```

```
} ePollThreadState;

/* ----------------------- Static functions ---------------------------------*/
static BOOL     bCreatePollingThread( void );
static enum ThreadState eGetPollingThreadState( void );
static void     eSetPollingThreadState( enum ThreadState eNewState );
static DWORD WINAPI dwPollingThread( LPVOID lpParameter );

/* ----------------------- Start implementation -----------------------------*/
int
_tmain( int argc, _TCHAR * argv[] )
{
    int             iExitCode;
    TCHAR           cCh;
    BOOL            bDoExit;

    if( eMBTCPInit( MB_TCP_PORT_USE_DEFAULT ) != MB_ENOERR )
    {
        _ftprintf( stderr, _T( "%s: can't initialize modbus stack!\r\n" ),
PROG );
        iExitCode = EXIT_FAILURE;
    }
    else
    {
        /* Create synchronization primitives and set the current state
         * of the thread to STOPPED.
         */
        InitializeCriticalSection( &hPollLock );
        eSetPollingThreadState( STOPPED );

        /* CLI interface. */
        _tprintf( _T( "Type 'q' for quit or 'h' for help!\r\n" ) );
        bDoExit = FALSE;
        do
        {
            _tprintf( _T( "> " ) );
            cCh = _gettchar(  );
            switch ( cCh )
            {
            case _TCHAR( 'q' ):
                bDoExit = TRUE;
                break;
            case _TCHAR( 'd' ):
                eSetPollingThreadState( SHUTDOWN );
                break;
            case _TCHAR( 'e' ):
                if( bCreatePollingThread(  ) != TRUE )
                {
                    _tprintf( _T( "Can't start protocol stack! Already running?
\r\n" ) );
                }
                break;
            case _TCHAR( 's' ):
```

```
                switch ( eGetPollingThreadState(  ) )
                {
                case RUNNING:
                    _tprintf( _T( "Protocol stack is running.\r\n" ) );
                    break;
                case STOPPED:
                    _tprintf( _T( "Protocol stack is stopped.\r\n" ) );
                    break;
                case SHUTDOWN:
                    _tprintf( _T( "Protocol stack is shuting down.\r\n" ) );
                    break;
                }
                break;
            case _TCHAR( 'h' ):
                _tprintf( _T( "FreeModbus demo application help:\r\n" ) );
                _tprintf( _T( "  'd' ... disable protocol stack.\r\n" ) );
                _tprintf( _T( "  'e' ... enabled the protocol stack\r\n" ) );
                _tprintf( _T( "  's' ... show current status\r\n" ) );
                _tprintf( _T( "  'q' ... quit applicationr\r\n" ) );
                _tprintf( _T( "  'h' ... this information\r\n" ) );
                _tprintf( _T( "\r\n" ) );
                _tprintf( _T( "Copyright 2006 Christian Walter <wolti@sil.at>\r
\n" ) );
                break;
            default:
                if( cCh != _TCHAR('\n') )
                {
                    _tprintf( _T( "illegal command '%c'!\r\n" ), cCh );
                }
                break;
            }

            /* eat up everything untill return character. */
            while( cCh != '\n' )
            {
                cCh = _gettchar(  );
            }
        }
        while( !bDoExit );

        /* Release hardware resources. */
        ( void )eMBClose(  );
        iExitCode = EXIT_SUCCESS;
    }
    return iExitCode;
}

BOOL
bCreatePollingThread( void )
{
    BOOL            bResult;

    if( eGetPollingThreadState(  ) == STOPPED )
```

```
    {
        if( ( hPollThread = CreateThread( NULL, 0, dwPollingThread, NULL, 0,
NULL ) ) == NULL )
        {
            /* Can't create the polling thread. */
            bResult = FALSE;
        }
        else
        {
            bResult = TRUE;
        }
    }
    else
    {
        bResult = FALSE;
    }

    return bResult;
}

DWORD           WINAPI
dwPollingThread( LPVOID lpParameter )
{
    eSetPollingThreadState( RUNNING );

    if( eMBEnable(  ) == MB_ENOERR )
    {
        do
        {
            if( eMBPoll(  ) != MB_ENOERR )
                break;
        }
        while( eGetPollingThreadState(  ) != SHUTDOWN );
    }

    ( void )eMBDisable(  );

    eSetPollingThreadState( STOPPED );

    return 0;
}

enum ThreadState
eGetPollingThreadState(  )
{
    enum ThreadState eCurState;

    EnterCriticalSection( &hPollLock );
    eCurState = ePollThreadState;
    LeaveCriticalSection( &hPollLock );

    return eCurState;
}
```

```
void
eSetPollingThreadState( enum ThreadState eNewState )
{
    EnterCriticalSection( &hPollLock );
    ePollThreadState = eNewState;
    LeaveCriticalSection( &hPollLock );
}

eMBErrorCode
eMBRegInputCB( UCHAR * pucRegBuffer, USHORT usAddress, USHORT usNRegs )
{
    eMBErrorCode    eStatus = MB_ENOERR;
    int             iRegIndex;

    if( ( usAddress >= REG_INPUT_START )
        && ( usAddress + usNRegs <= REG_INPUT_START + REG_INPUT_NREGS ) )
    {
        iRegIndex = ( int )( usAddress - usRegInputStart );
        while( usNRegs > 0 )
        {
            *pucRegBuffer++ = ( unsigned char )( usRegInputBuf[iRegIndex] >>
8 );
            *pucRegBuffer++ = ( unsigned char )( usRegInputBuf[iRegIndex] &
0xFF );
            iRegIndex++;
            usNRegs--;
        }
    }
    else
    {
        eStatus = MB_ENOREG;
    }

    return eStatus;
}

eMBErrorCode
eMBRegHoldingCB( UCHAR * pucRegBuffer, USHORT usAddress, USHORT usNRegs,
eMBRegisterMode eMode )
{
    eMBErrorCode    eStatus = MB_ENOERR;
    int             iRegIndex;

    if( ( usAddress >= REG_HOLDING_START ) &&
        ( usAddress + usNRegs <= REG_HOLDING_START + REG_HOLDING_NREGS ) )
    {
        iRegIndex = ( int )( usAddress - usRegHoldingStart );
        switch ( eMode )
        {
            /* Pass current register values to the protocol stack. */
        case MB_REG_READ:
```

```
            while( usNRegs > 0 )
            {
                *pucRegBuffer++ = ( UCHAR ) ( usRegHoldingBuf[iRegIndex] >> 8 );
                *pucRegBuffer++ = ( UCHAR ) ( usRegHoldingBuf[iRegIndex] &
0xFF );
                iRegIndex++;
                usNRegs--;
            }
            break;

            /* Update current register values with new values from the
             * protocol stack. */
        case MB_REG_WRITE:
            while( usNRegs > 0 )
            {
                usRegHoldingBuf[iRegIndex] = *pucRegBuffer++ << 8;
                usRegHoldingBuf[iRegIndex] |= *pucRegBuffer++;
                iRegIndex++;
                usNRegs--;
            }
        }
    }
    else
    {
        eStatus = MB_ENOREG;
    }
    return eStatus;
}


eMBErrorCode
eMBRegCoilsCB( UCHAR * pucRegBuffer, USHORT usAddress, USHORT usNCoils,
eMBRegisterMode eMode )
{
    return MB_ENOREG;
}

eMBErrorCode
eMBRegDiscreteCB( UCHAR * pucRegBuffer, USHORT usAddress, USHORT usNDiscrete )
{
    return MB_ENOREG;
}
```

FreeModbus Related Pages

Here is a list of all related documentation pages:

- **Porting for RTU/ASCII**
- **Tips**

Automatically generated by Doxygen 1.3.9.1 on 19 Nov 2006.

Porting for RTU/ASCII

The first steps should always be to create a new directory for the port. The recommended layout is to create a top level directory, e.g. `demo/PLATFORM` which hold the application and project files. In addition a subdirectory `port` should be created for the port specific files.

```
demo/PLATFORM/Makefile
demo/PLATFORM/main.c
demo/PLATFORM/port/portserial.c
demo/PLATFORM/port/porttimer.c
demo/PLATFORM/port/portother.c
demo/PLATFORM/port/port.h
```

You can use `demo/BARE` as a starting point. Simply copy the directory and rename it to a name of your choice.

Platform specifics (port.h)

You should first check the file **`port.h`** and check the if the examples are already suitable for your platform. You must at least define the macros for enabling `ENTER_CRITICAL_SECTION` and disabling `EXIT_CRITICAL_SECTION` interrupts.

Implementation of the timer functions (porttimer.c)

The Modbus protocol stacks needs a timer to detect the end of the frame. The timers should have a resolution of half the time of a serial character. For example for 38400 baud the character time is approx. 280us assuming 11bits for a single character. The smallest timeout used by the protocol stack is 3.5 times the character timeout.

You should start by implementing the function `xMBPortTimersInit( USHORT usTim1Timerout50us )` and `vMBPortTimersEnable( )`. Test the function with the following sample code:

```
vMBPortTimersInit( 20 );
vMBPortTimersEnable( );
for( ;; );
```

Place a breakpoint or toggle an LED in the interrupt handler which calls `pxMBPortCBTimerExpired`. The ISR should occur approx. 1ms after the call to `vMBPortTimersEnable()`. You should also check that `vMBPortTimersDisable( )` works as expected.

Note:

If you use Modbus ASCII the timers are in the range of seconds because the timeouts are much larger there. Make sure you can handle a value of 20000 for `usTim1Timerout50us`

which corresponds to an one second timeout. See **mbconfig.h** for the value of the timeout defined by MB_ASCII_TIMEOUT_SEC.

Porting for RTU/ASCII

The serial porting layer must be capable of initializing the UART, disabling and enabling the receiver and transmitter components as well as performing callbacks if a character has been received or can be transmitted. You should start by implementing `xMBPortSerialInit( UCHAR ucPORT, ULONG ulBaudRate, UCHAR ucDataBits, eMBParity eParity )` and `vMBPortSerialEnable ( BOOL xRxEnable, BOOL xTxEnable )`. In addition you need to create two interrupt service routines for you communication devices. It is usually simpler to start with the receive interrupt.

Create an interrupt handler for the receive interrupt, set a breakpoint there and check if `xMBPortSerialGetByte( CHAR * pucByte )` correctly returns the character. This can be tested by the following code:

```
/* Initialize COM device 0 with 38400 baud, 8 data bits and no parity. */
if( xMBPortSerialInit( 0, 38400, 8, MB_PAR_NONE ) == FALSE )
{
  fprintf(stderr, "error: com init failed");
}
else
{
  /* Enable the receiver. */
  vMBPortSerialEnable( TRUE, FALSE );
  /* Now block. Any character received should cause an interrupt now. */
  for( ;; );
}
```

And your serial character received ISR should look like:

```
static void prvvUARTTxReadyISR( void )
{
    CHAR cByte;
    ( void )xMBPortSerialGetByte( &cByte );
    /* Now cByte should contain the character received. */
}
```

Next you should check that the transmitter part is actually working as expected. Open a terminal program and simply call `xMBPortSerialPutByte( 'a' )` in the transmit buffer empty ISR. If you use the sample code from below exactly 10 characters should be received.

```
/* Initialize COM device 0 with 38400 baud, 8 data bits and no parity. */
if( xMBPortSerialInit( 0, 38400, 8, MB_PAR_NONE ) == FALSE )
{
  fprintf(stderr, "error: com init failed");
}
else
```

```
{
  /* Enable the transmitter. */
  vMBPortSerialEnable( FALSE, TRUE );
  /* Now block. Any character received should cause an interrupt now. */
  for( ;; );
}
```

And you serial transmit buffer empty ISR should look like:

```
static unsigned int uiCnt = 0;

void prvvUARTTxReadyISR( void )
{
    if( uiCnt++ < 10 )
    {
        ( void )xMBPortSerialPutByte( 'a' );
    }
    else
    {
        vMBPortSerialDisable( FALSE, FALSE );
    }
}
```

If you are sure everything works correctly change the interrupt routines back to the examples shown in **`portserial.c`**

Implementing the event queue (portevent.c)

If you are not using an operating system the port is already finished and the demo application should work as expected. If you in the luck of having an operating system usage of the FreeModbus protocol stack differs in the following way:

- Create another task at startup which calls **eMBPoll( )** in a loop. This should look like:

  ```
  for( ;; )
  {
      ( void )eMBPoll(  );
  }
  ```

  See the STR71x port for an FreeRTOS example.

- Change the function `xMBPortEventPost` to post an event to a queue. Note that this function will be called from an ISR so check your RTOS documentation for that.

- Change the `xMBPortEventGet` to retrieve an event from that queue. The function `eMBPoll` periodically calls it. The function should block until an event has been posted to the queue.

In addition the serial and timer interrupt function must be modified. Whenever the protocol handler callback functions `pxMBFrameCBByteReceived`, `pxMBFrameCBTransmitterEmpty` and

`pxMBPortCBTimerExpired` return `TRUE` a context switch should be made after exiting the ISR because an event has been posted to the queue. Forgetting to do this will result in slow performance of the protocol stack.

Automatically generated by Doxygen 1.3.9.1 on 19 Nov 2006.

Tips

This page provides some tips for using the FreeModbus protocol stack.

Reducing memory requirements

The memory requirements of FreeModbus can be tuned in the following way. These are basic tricks and can easily be done:

- Decided if you need RTU, ASCII and TCP at the same time. If not disable them in the file **mbconfig.h** by settings the respective options MB_RTU_ENABLED, MB_ASCII_ENABLED and MB_TCP_ENABLED to zero.
- If you don't need all Modbus functions disable them in the file **mbconfig.h**. This will reduce code requirements.
- Set the variable MB_FUNC_HANDLERS_MAX in **mbconfig.h** to the number of functions codes you want to support.

If you have stronger limits you can also try the following options. Note that this options have an impact on the features of the protocol stack.

- Use some compiler directive to put the mapping of function codes to handler functions into the flash memory of you CPU. You can find this table in the file mb.c at the top of the file. The static variable is named xFuncHandlers.

- Reduce the size of the RTU buffer. In this case longer frames will result in an error (Your device will drop all these frames). This is possible if you will never get read/write requests with that number of registers or your total amount of registers is small anyway.

- You could also remove some function pointers which make the protocol stack configurable and replace them by the functions itself. For example if you only want to use RTU remove the callback functions from the porting layer and fill in the appropriate calls. This will save the space for all function pointers.

Automatically generated by Doxygen 1.3.9.1 on 19 Nov 2006.