M032 FreeMODBUS Slave RTU Mode Porting

Example Code Introduction for 32-bit NuMicro® Family

Document Information

Application	This example code demonstrates how to port the FreeMODBUS slave RTU mode to the M032 series microcontrollers.
BSP Version	M031_Series_BSP_CMSIS_V3.06.000
Hardware	NuMaker-M032SE V1.3

The information described in this document is the exclusive intellectual property of Nuvoton Technology Corporation and shall not be reproduced without permission from Nuvoton.

Nuvoton is providing this document only for reference purposes of NuMicro microcontroller and microprocessor based system design. Nuvoton assumes no responsibility for errors or omissions.

All data and specifications are subject to change without notice.

For additional information or questions, please contact: Nuvoton Technology Corporation.

www.nuvoton.com

1. Overview

This example code demonstrates how to use the UART interface and implement a MODBUS slave in RTU mode. MODBUS RTU is an open serial communication protocol with a master/slave structure. MODBUS RTU message encoding is hexadecimal and includes CRC checksum packets.

This code ports FreeMODBUS to implement the MODBUS slave. FreeMODBUS is a free, universal MODBUS protocol library designed for embedded applications. FreeMODBUS supports RTU, ASCII transmission modes, and TCP protocols. FreeMODBUS follows the BSD license, which means users can apply FreeMODBUS in commercial environments.

Users can download the FreeMODBUS source code from the following official website link:

FreeMODBUS Downloads - Embedded Experts (embedded-experts.at)

1.1 Principle

The MODBUS protocol is a master/slave communication protocol. Each slave must have a unique address, with a maximum of 247 devices. MODBUS communication interfaces can be RS232, RS485, or Ethernet. The communication follows these steps:

1. The master device sends a request to the slave device.
2. The slave device analyzes and processes the master device's request, then sends the result back to the master device.
3. If any errors occur, the slave device returns an exception function code.

There are three transmission modes in MODBUS:

- ASCII (American Standard Code for Information Interchange).
- RTU (Remote Terminal Unit).
- TCP (Transmission Control Protocol).

This example uses the MODBUS RTU mode, which employs hexadecimal encoding and a 2 bytes CRC checksum. Each 8-bit byte in the message contains two 4-bit hexadecimal characters. At the same serial transmission rate, MODBUS RTU mode can transmit more data compared to ASCII mode. The time interval for sending or receiving between any two bytes must not exceed 3.5 times the character transmission time.

The MODBUS RTU data frame consists of four parts: device address, function code, data, and checksum, as shown in the following Table 1-1.

Slave Address	Function Code	Data (Follow function code)	CRC
1 Byte	1 Byte	0 ~ 252 Bytes	2 Bytes

Table 1-1 MODBUS RTU Data Frame

Common function codes are listed in Table 1-2:

Function Code	**Name**	**Explanation**
0x1	Read coil Status	Retrieve the current status of a group of digital outputs
0x2	Read Input Status	Retrieve the current status of a set of digital inputs
0x3	Read Holding Registers	Retrieve the current status of a set of analog outputs
0x4	Read Input Registers	Retrieve the current status of a set of analog inputs
0x5	Force Single Coil	Forcefully set the value of a single digital output
0x6	Preset Single Register	Forcefully set the value of a single analog output
0xF	Force Multiple Coils	Forcefully set the values of multiple digital outputs
0x10	Preset Multiple Registers	Forcefully set the values of multiple analog outputs

Table 1-2 MODBUS RTU Common Function Codes

The slave address/slave ID is the device's ID. When reading or writing to a slave, it's necessary to know the address/ID of that device. The commonly used MODBUS function code is 0x03. In the 0x03 function code, the defined function is to read multiple registers, used to read data from consecutive addresses. Below is an example of MODBUS RTU communication:

MODBUS RTU master read format:

Device ID (slave address/ID) + 0x03 + Starting address to read (2 bytes) + Quantity to read (2 bytes) + CRC16.

MODBUS RTU slave response format:

Device ID (slave address/ID) + 0x03 + Number of bytes in response data + Data 1 (1 byte) + ... + Data n (1 byte) + CRC16.

1.2 Demo Result

To test the MODBUS protocol, you can download Modpoll and use your personal computer (PC) as the master controller. Connect the PC to the NuMaker-M032SE board using NU-LINK2-VCOM and connect it to M032's UART for testing.

You can find Modpoll on the official website and download it from the following link:

Free Modbus Master Simulator and Test Tool (modbusdriver.com)

First, download and run the FreeMODBUS porting code for M032. Then, in the Modpoll folder, execute the following command in the command line on your PC:

modpoll.exe -m rtu -a 10 -r 1000 -c 4 -t 3 -b 38400 -d 8 -p none COM[n, serial port number]

For the modpoll parameters, please refer to the instructions on the official website. Replace COM[n] with the actual serial port used on your PC.

```
\modpoll.exe -m rtu -a 10 -r 1000 -c 4 -t 3 -b 38400 -d 8 -p none  COM31
modpoll 3.4 - FieldTalk(tm) Modbus(R) Master Simulator
Copyright (c) 2002-2013 proconX Pty Ltd
Visit http://www.modbusdriver.com for Modbus libraries and tools.

Protocol configuration: Modbus RTU
Slave configuration...: address = 10, start reference = 1000, count = 4
Communication.........: COM31, 38400, 8, 1, none, t/o 1.00 s, poll rate 1000 ms
Data type.............: 16-bit register, input register table

-- Polling slave... (Ctrl-C to stop)
[1000]: 16513
[1001]: 0
[1002]: 0
[1003]: 0
-- Polling slave... (Ctrl-C to stop)
[1000]: 12967
[1001]: 0
[1002]: 0
[1003]: 0
-- Polling slave... (Ctrl-C to stop)
[1000]: 5544
[1001]: 0
[1002]: 0
[1003]: 0
```

Figure 1-1 Executing Modpoll for Communication

2. Code Description

The porting of FreeMODBUS to the M032 requires that the chip functionalities of UART and TIMER be available. In the portserial.c code, the UART functionality must be ported using vMBPortSerialEnable, vMBPortClose, xMBPortSerialInit, xMBPortSerialPutByte, xMBPortSerialGetByte and UART02_IRQHandler. An example code is as follows:

```
/* ----------------------- Start implementation -----------------------------*/
/**
  * @brief      vMBPortSerialEnable
  * @param[in]  BOOL xRxEnable, BOOL xTxEnable
  * @return     None
  * @details    This function to enable/disabele UART TX and RX interrupt.
  */
void  vMBPortSerialEnable(BOOL xRxEnable, BOOL xTxEnable)
{
    __disable_irq();

    if (xRxEnable)
    {
        UART_ENABLE_INT(UART0, UART_INTEN_RDAIEN_Msk);
    }
    else
    {
        UART_DISABLE_INT(UART0, UART_INTEN_RDAIEN_Msk);
    }
    if (xTxEnable)
    {
        UART_ENABLE_INT(UART0, UART_INTEN_THREIEN_Msk);
    }
    else
    {
        UART_DISABLE_INT(UART0, UART_INTEN_THREIEN_Msk);
    }
    NVIC_EnableIRQ(UART02_IRQn);
    __enable_irq();
}

/**
  * @brief      vMBPortClose
  * @param[in]  None
  * @return     None
  * @details    This function to Close UART port.
  */
void vMBPortClose(void)
{
    /* Disable RDA and THRE Interrupt */
    UART_DISABLE_INT(UART0, UART_INTEN_RDAIEN_Msk | UART_INTEN_THREIEN_Msk);
    NVIC_DisableIRQ(UART02_IRQn);
}

/**
  * @brief      xMBPortSerialInit
  * @param[in]  UCHAR ucPORT, ULONG ulBaudRate, UCHAR ucDataBits, eMBParity eParity
  * @return     TURE or FALSE
  * @details    This function to Open UART port and set ulBaudRate, ucDataBits, eParity.
  */
BOOL xMBPortSerialInit(UCHAR ucPORT, ULONG ulBaudRate, UCHAR ucDataBits, eMBParity
eParity)
{
    if (ucPORT != 0) return FALSE;

    /* Step 1. GPIO initial */
```

```
    /* Set PB multi-function pins for UART0 RXD=PB.12 and TXD=PB.13 */
    SYS->GPB_MFPH = (SYS->GPB_MFPH & ~(SYS_GPB_MFPH_PB12MFP_Msk|
                     SYS_GPB_MFPH_PB13MFP_Msk))|\
                    (SYS_GPB_MFPH_PB12MFP_UART0_RXD | SYS_GPB_MFPH_PB13MFP_UART0_TXD);

    /* Step 2. Enable and Select UART clock source*/
    /* Enable UART0 clock */
    CLK_EnableModuleClock(UART0_MODULE);

    /* UART0 clock source */
    /* Select UART module clock source as HIRC and UART module clock divider as 1 */
    CLK_SetModuleClock(UART0_MODULE, CLK_CLKSEL1_UART0SEL_HIRC, CLK_CLKDIV0_UART0(1));

    /* Step 3. Select Operation BaudRate */
    SYS_ResetModule(UART0_RST);
    UART_Open(UART0, ulBaudRate);

    /* Step 4. Set Configure parity*/
    switch (eParity)
    {
        case MB_PAR_NONE:
            UART0->LINE &= ~UART_PARITY_EVEN;
            break;

        case MB_PAR_EVEN:
            UART0->LINE &= ~UART_PARITY_EVEN;
            UART0->LINE |= UART_PARITY_EVEN;
            break;

        case MB_PAR_ODD:
            UART0->LINE &= ~UART_PARITY_EVEN;
            UART0->LINE |= UART_PARITY_ODD;
            break;
    }

    /* Step 5. Set Configure data length*/
    UART0->LINE &= ~UART_WORD_LEN_8;

    switch (ucDataBits)
    {
        case 5:
            UART0->LINE |= UART_WORD_LEN_5;      //5 bits Data Length
            break;

        case 6:
            UART0->LINE |= UART_WORD_LEN_6;      //6 bits Data Length
            break;

        case 7:
            UART0->LINE |= UART_WORD_LEN_7;      //7 bits Data Length
            break;

        case 8:
            UART0->LINE |= UART_WORD_LEN_8;      //8 bits Data Length
            break;
    }

    return TRUE;
}

/**
  * @brief      xMBPortSerialPutByte
  * @param[in]  CHAR ucByte
  * @return     TURE or FALSE
  * @details    This function to send ucByte to UART TX buffer.
  */
BOOL xMBPortSerialPutByte(CHAR ucByte)
```

```
{

    while (UART_GET_TX_FULL(UART0) != 0x00); //check Tx Fifo full.

    UART0->DAT = (uint8_t) ucByte;

    return TRUE;
}
/**
  * @brief      xMBPortSerialGetByte
  * @param[in]  CHAR *pucByte
  * @return     TURE or FALSE
  * @details    This function to get pucByte from UART RX buffer.
  */
BOOL xMBPortSerialGetByte(CHAR *pucByte)
{

    while (UART_GET_RX_EMPTY(UART0) != 0x00); //check Rx Empty

    *pucByte = UART0->DAT;

    return TRUE;
}

/**
  * @brief      UART02_IRQHandler
  * @param[in]  None
  * @return     None
  * @details    This function is UART TX and RX interrupt handler function.
  */
void UART02_IRQHandler(void)
{

    if (UART_GET_INT_FLAG(UART0, UART_INTSTS_RDAINT_Msk))
    {

        /* Get all the input characters */
        while (UART_IS_RX_READY(UART0))
        {
         pxMBFrameCBByteReceived();
        }

    }
    else if (UART_GET_INT_FLAG(UART0, UART_INTSTS_THREINT_Msk))
    {
    pxMBFrameCBTransmitterEmpty();
    }
}
```

In the porttimer.c code, utilizing the TIMER requires porting the following functions: xMBPortTimersInit, vMBPortTimersEnable, vMBPortTimersDisable, and TMR0_IRQHandler.

```
/**
  * @brief      xMBPortTimersInit
  * @param[in]  USHORT usTim1Timerout50us
  * @return     TURE or FALSE
  * @details    This function to Open Timer0 and time out count(50us unit).
  */
BOOL xMBPortTimersInit(USHORT usTim1Timerout50us)
{
    float freq_exp = 0;
    freq_exp = 1.0 / ((float)usTim1Timerout50us * 0.00005);//convert time to frequence
```

```
    /* Enable Timer0 clock source */
    CLK_EnableModuleClock(TMR0_MODULE);

    /* Select Timer0 clock source as internal 48M */
    CLK_SetModuleClock(TMR0_MODULE, CLK_CLKSEL1_TMR0SEL_HIRC, 0);

    /* Reset IP TMR0 */
    SYS_ResetModule(TMR0_RST);

    /* Open Timer0 in periodic mode, enable interrupt and interrupt tick per second */
    TIMER_Open(TIMER0, TIMER_PERIODIC_MODE, (uint32_t)freq_exp);

    /* Enable Timer0 interrupt */
    TIMER_EnableInt(TIMER0);

    /* Enable Timer0 NVIC */
    NVIC_EnableIRQ(TMR0_IRQn);
    return TRUE;
}

/**
  * @brief      vMBPortTimersEnable
  * @param[in]  None
  * @return     None
  * @details    This function to enable Timer0 and time count reset.
  */
void vMBPortTimersEnable(void)
{
    /* Reset Timer0 counter */
    TIMER_ResetCounter(TIMER0);
    /* Enable Timer0 */
    TIMER_Start(TIMER0);
}

/**
  * @brief      vMBPortTimersDisable
  * @param[in]  None
  * @return     None
  * @details    This function to disable Timer0 and time count reset.
  */
void vMBPortTimersDisable(void)
{
    /* Disable Timer0 */
    TIMER_Stop(TIMER0);
    /* Reset Timer0 counter */
    TIMER_ResetCounter(TIMER0);
}

/**
  * @brief      TMR0_IRQHandler
  * @param[in]  None
  * @return     None
  * @details    This function is Timer0 interrupt handler
  */
void TMR0_IRQHandler(void)
{
```

```
    /* Clear Timer0 interrupt flag */
    TIMER_ClearIntFlag(TIMER0);
    (void)pxMBPortCBTimerExpired();
}
```

In the main function, call eMBInit to handle setting up RTU mode, configuring the slave address, opening and setting up UART with the desired baud rate and parity settings, and opening the MODBUS protocol stack. Finally, within the main loop, handle eMBPoll to determine MODBUS packet events and process them.

```
int32_t main(void)
{
    …
    /* Enable the MODBUS RTU mode, MODBUS slave Address, UART port, BaudRate, Parity*/
    eMBInit(MB_RTU, 0x0A, 0, 38400, MB_PAR_NONE);

    /* Enable the MODBUS Protocol Stack. */
    eMBEnable();

    for (;;)
    {
        /*eMBPoll for MODBUS event process*/
        (void)eMBPoll();

        /* Here we simply count the number of poll cycles. */
        /* The usRegInputBuf[0] will be accumulated in the main loop, allowing the master
        to read the data changes.*/
        usRegInputBuf[0]++;
    }
}
```

3. Software and Hardware Requirements

3.1 Software Requirements

- BSP version
  - M032_Series_BSP_CMSIS_V3.06.000
- IDE version
  - Keil uVersion 5.28

3.2 Hardware Requirements

- Circuit components
  - NuMaker-M032SE V1.3
- Pin Connect
  - Enable VCOM on the NuMaker-M032SE's Nu-Link2-Me board by toggling SW1 and SW2 to the ON position. This allows communication between the host and M032 as part of the MODBUS protocol.

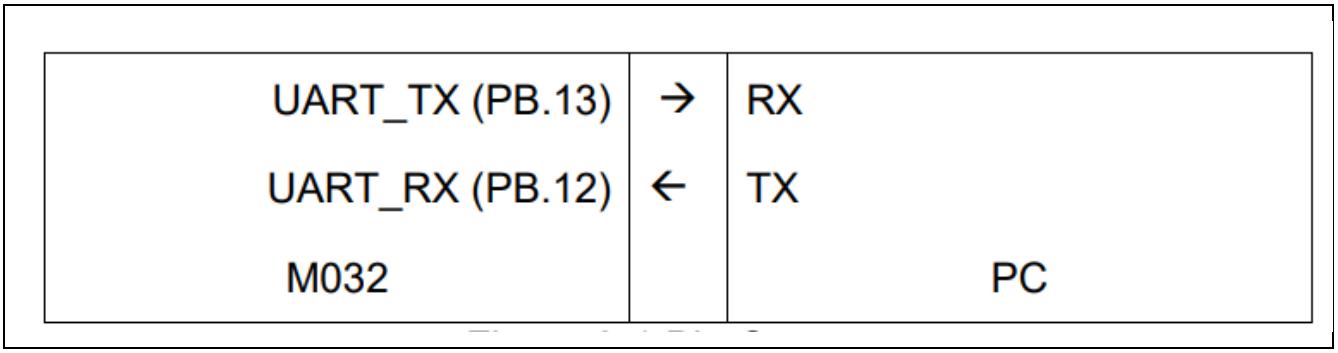

Figure 3-1 Pin Connect

4. Directory Information

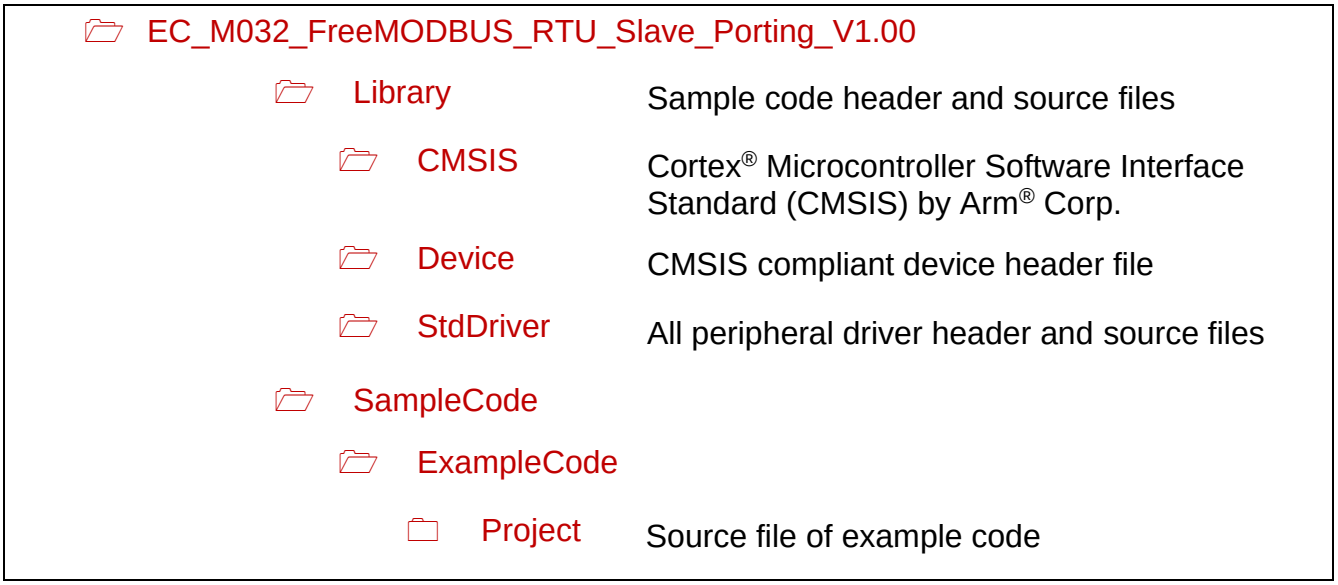

Figure 4-1 Directory Structure

5. Example Code Execution

1. Browse the sample code folder as described in the Directory Information section and double-click *FreeMODBUS_RTU_Slave_Porting.uvprojx*.
2. Enter Keil compile mode.
   - Build
   - Download
   - Start/Stop debug session
3. Enter debug mode.
   - Run

6. Revision History

Date	Revision	Description
2024.05.20	1.00	Initial version.

Important Notice

Nuvoton Products are neither intended nor warranted for usage in systems or equipment, any malfunction or failure of which may cause loss of human life, bodily injury or severe property damage. Such applications are deemed, "Insecure Usage".

Insecure usage includes, but is not limited to: equipment for surgical implementation, atomic energy control instruments, airplane or spaceship instruments, the control or operation of dynamic, brake or safety systems designed for vehicular use, traffic signal instruments, all types of safety devices, and other applications intended to support or sustain life.

All Insecure Usage shall be made at customer's risk, and in the event that third parties lay claims to Nuvoton as a result of customer's Insecure Usage, customer shall indemnify the damages and liabilities thus incurred by Nuvoton.

Please note that all data and specifications are subject to change without notice.
All the trademarks of products and companies mentioned in this datasheet belong to their respective owners.