Using LwIP to Implement Modbus TCP

Example Code Introduction for 32-bit NuMicro® Family

Document Information

Application	This example code uses the LwIP library to implement the Modbus TCP function for industrial network application. User can control the status of M467 through an industrial network server by the Modbus TCP protocol.
BSP Version	M460 Series BSP CMSIS V3.00.001
Hardware	NuMaker-M467HJ V1.0

1. Overview

This example code uses the LwIP library to implement the Modbus TCP function for industrial network application. User can control the status of the M467 series microcontroller (MCU) through an industrial network server by the Modbus TCP protocol.

Modbus is an industrial communication protocol used for system monitoring and remote data collection in automatic systems. Common communication interfaces for Modbus include RS-232, RS-485, Ethernet, etc. It defines a set of commands and data formats that allow a master device to send instructions to slave devices, which can be used for reading and writing slave data.

Modbus is also used for monitoring and controlling various devices like programmable logic controllers (PLCs), sensors, valves, servos, human-machine interfaces (HMIs), etc. It provides a standardized communication method for devices from different vendors to communicate with each other. Modbus protocol enhance the efficiency and reliability of automatic systems.

1.1 Principle

The Modbus protocol defines multiple control instructions for controlling the behavior of slave devices. The following are commonly used control instructions in the Modbus protocol:

Command Code	Command Name
01	Read Coils
02	Read Discrete Inputs
03	Read Holding Registers
04	Read Input Register
05	Write Single Coil
06	Write Single Register
15	Write Multiple Coils
16	Write Multiple Registers

Table 1-1 Command Code

The following describes how the master controls the slave devices by the control instructions:

- **Read Coils**: This instruction is used to read a set of binary data (typically representing status) from the slave.
- **Read Discrete Inputs**: Similar to Read Coils, this instruction is primarily used to read the status of discrete inputs, which typically represent the status of switches or buttons.
- **Read Holding Registers**: This instruction is used to read 16-bit numeric data from the slave's holding registers.
- **Read Input Register**: Similar to Read Holding Registers, this instruction is used to read the values of input register. Input register is typically used to store sensor data.
- **Write Single Coil**: This instruction is used to write a single binary data (bit) to the slave's coil.

- **Write Single Register**: This instruction is used to write a single 16-bit numeric data to a register in the slave.
- **Write Multiple Coils**: This instruction is used to write multiple binary data (bits) to different coils in the slave.
- **Write Multiple Registers**: This instruction is used to write multiple 16-bit numeric data to different registers in the slave.

These control instructions provide basic control functionalities between the master and slave devices. Modbus is commonly chosen for monitoring and controlling various devices in an automatic system. The master can use these instructions to perform read or write operations as needed, thereby achieving control and adjustment of the slave devices.

1.2 Demo Result

After compiling and programming the example code, user could download and use the Modbus TCP PC tool Modbus Poll to test this example code.

To perform Modbus Poll operations, follow the steps below.

1. Click "**Connect**" icon. Set the IP address to 192.168.1.2, ensuring that it is in the same network domain as the PC.

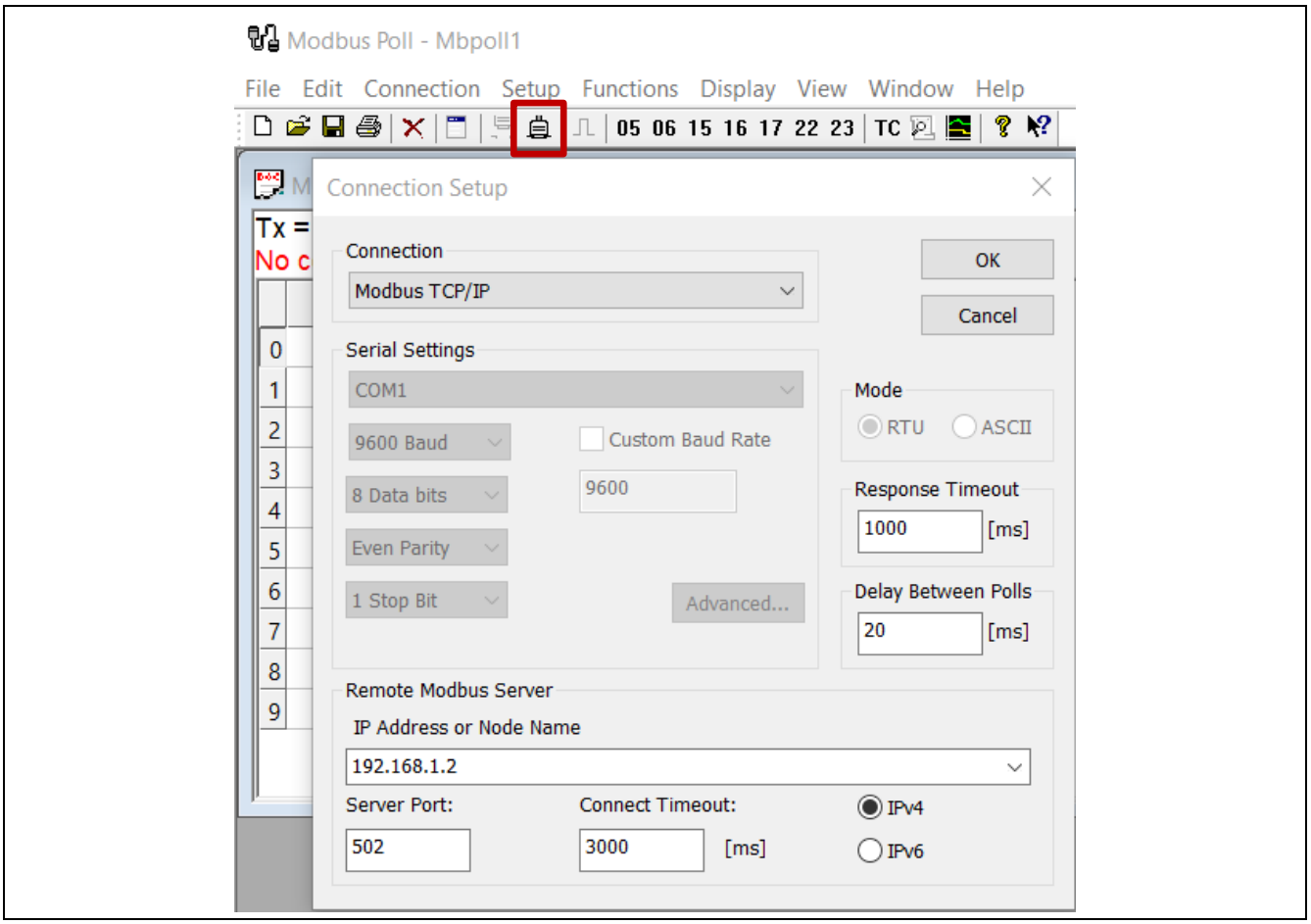

Figure 1-1 Connection Setup

2. Double click the value field of field 0 to open the command window. Set the value to 151 and click "**Send**" to transmit the data.

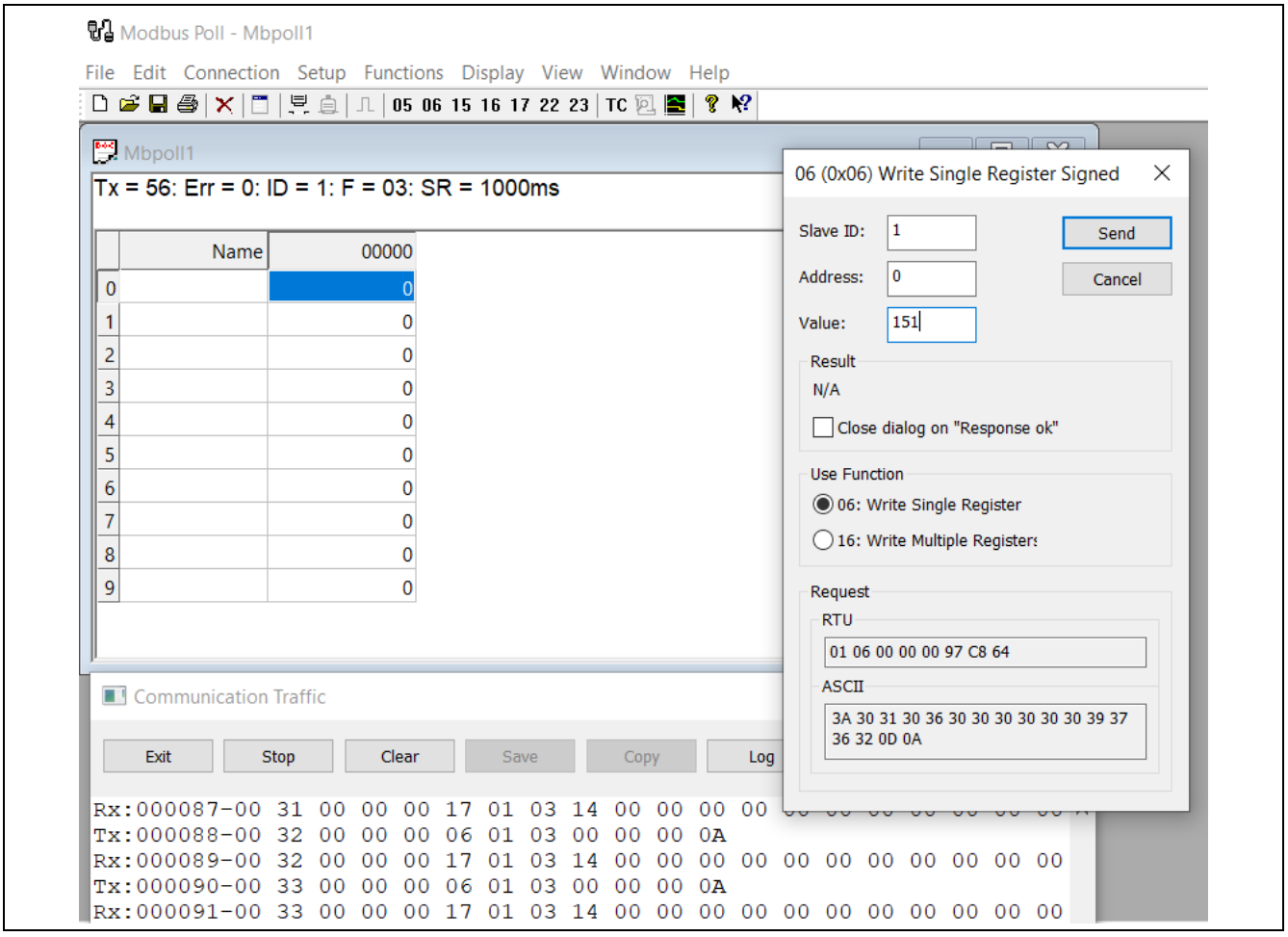

Figure 1-2 Send Data

The demo result is shown in Figure 1-3.

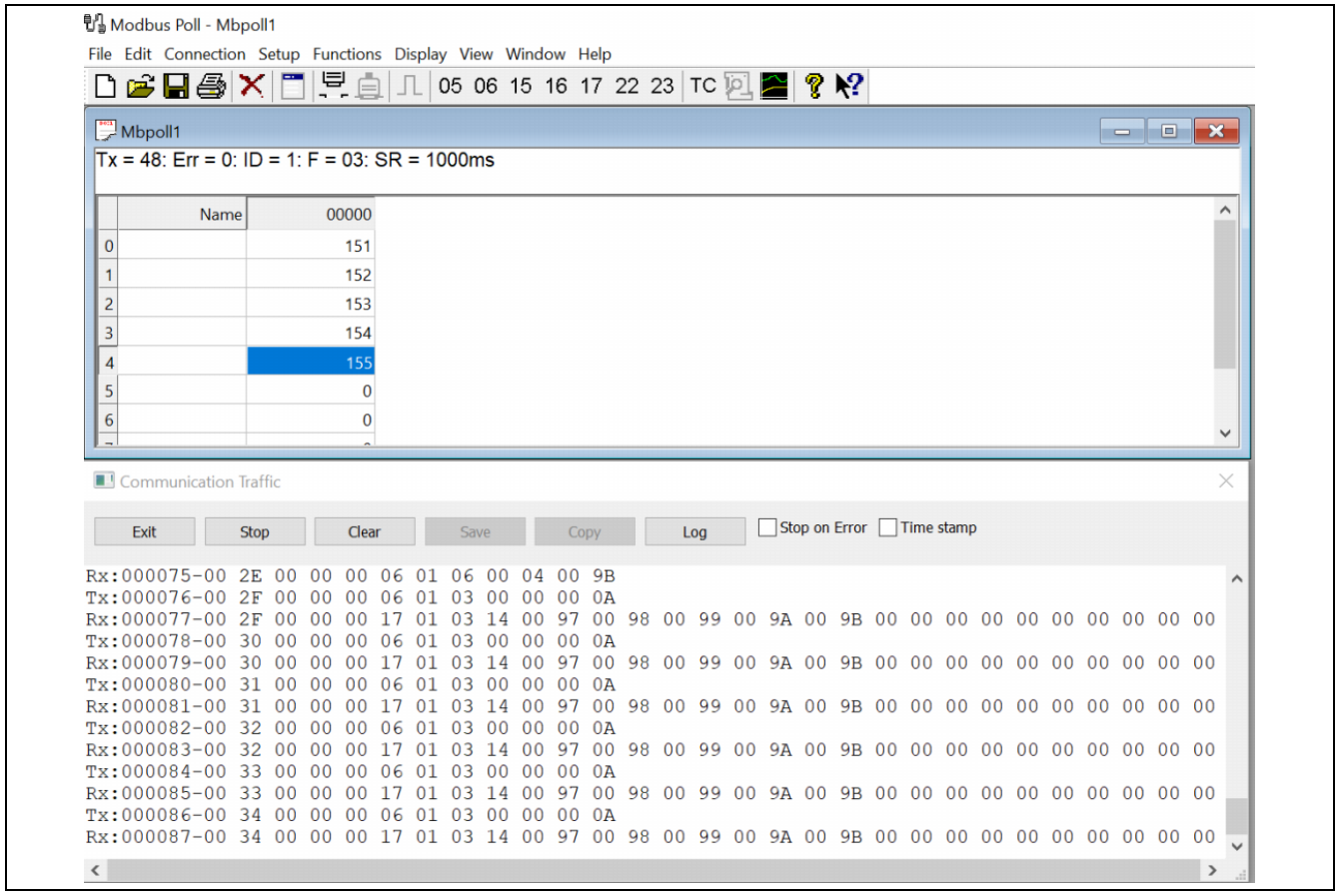

Figure 1-3 Modbus TCP Communication Data

2. Code Description

Configure the MCU hardware settings and create a TCP task on FreeRTOS scheduler to execute the network application. The code is located in *main.c*.

```
int main(void)
{
    /* Configure the hardware ready to run the test. */
    prvSetupHardware();

    xTaskCreate( vTcpTask, "TcpTask", TCPIP_THREAD_STACKSIZE, NULL,
mainCHECK_TASK_PRIORITY, NULL );

    printf("FreeRTOS is starting ...\n");

    /* Start the scheduler. */
    vTaskStartScheduler();

    /* If all is well, the scheduler will now be running, and the following line
    will never be reached.  If the following line does execute, then there was
    insufficient FreeRTOS heap memory available for the idle and/or timer tasks
    to be created.  See the memory management section on the FreeRTOS web site
    for more details. */
    for( ;; );
}
```

After initializing the system, execute the TCP task to configure parameters like the network IP, and start the Modbus task.

```
static void vTcpTask( void *pvParameters )
{
    ip_addr_t ipaddr;
    ip_addr_t netmask;
    ip_addr_t gw;

    IP4_ADDR(&gw, 192, 168, 1, 1);          //Set gateway IP 192.168.1.1
    IP4_ADDR(&ipaddr, 192, 168, 1, 2);      //Set IP address 192.168.1.2
    IP4_ADDR(&netmask, 255, 255, 255, 0);   //Set net mask 255.255.255.0

    tcpip_init(NULL, NULL);

    netif_add(&netif, &ipaddr, &netmask, &gw, NULL, ethernetif_init, tcpip_input);

    netif_set_default(&netif);
    netif_set_up(&netif);

    printf("[ TCP_Modbus ] \n");
    printf("IP address:      %s\n", ip4addr_ntoa(&netif.ip_addr));
    printf("Subnet mask:     %s\n", ip4addr_ntoa(&netif.netmask));
    printf("Default gateway: %s\n", ip4addr_ntoa(&netif.gw));

    tcp_netconn_init();

    vTaskSuspend( NULL );
```

```
}

static void tcp_netconn_thread(void *arg)
{
    eMBErrorCode    xStatus;

    for( ;; )
    {
        //Initialize Modbus stack
        if( eMBTCPInit( MB_TCP_PORT_USE_DEFAULT ) != MB_ENOERR )
        {
            printf("can't initialize modbus stack!\r\n");
        }
        else if( eMBEnable(  ) != MB_ENOERR )   //Enable Modbus stack
        {
            printf("can't enable modbus stack!\r\n");
        }
        else
        {
            do
            {
                xStatus = eMBPoll(  );      //Polling Modbus event
            }
            while( xStatus == MB_ENOERR );
        }
        /* An error occured. Maybe we can restart. */
        ( void )eMBDisable(  );
        ( void )eMBClose(  );
    }
}
```

The following introduces the instruction handling routines provided by the FreeModbus library:

- Callback function for coils:

The callback function handles the instructions associated with the coils.

```
eMBErrorCode eMBRegCoilsCB(UCHAR * pucRegBuffer, USHORT usAddress,
        USHORT usNCoils, eMBRegisterMode eMode)
{
    eMBErrorCode    eStatus = MB_ENOERR;
    USHORT          iRegIndex , iRegBitIndex , iNReg;
    UCHAR *         pucCoilBuf;
    USHORT          COIL_START;
    USHORT          COIL_NCOILS;
    USHORT          usCoilStart;
    iNReg =  usNCoils / 8 + 1;

    pucCoilBuf = ucSCoilBuf;
    COIL_START = S_COIL_START;
    COIL_NCOILS = S_COIL_NCOILS;
    usCoilStart = usSCoilStart;

    /* it already plus one in modbus function method. */
    usAddress--;
```

```
    if( ( usAddress >= COIL_START ) &&
        ( usAddress + usNCoils <= COIL_START + COIL_NCOILS ) )
    {
        iRegIndex = (USHORT) (usAddress - usCoilStart) / 8;
        iRegBitIndex = (USHORT) (usAddress - usCoilStart) % 8;
        switch ( eMode )
        {
        /* read current coil values from the protocol stack. */
        case MB_REG_READ:
            while (iNReg > 0)
            {
                *pucRegBuffer++ = xMBUtilGetBits(&pucCoilBuf[iRegIndex++],
                        iRegBitIndex, 8);
                iNReg--;
            }
            pucRegBuffer--;
            /* last coils */
            usNCoils = usNCoils % 8;
            /* filling zero to high bit */
            *pucRegBuffer = *pucRegBuffer << (8 - usNCoils);
            *pucRegBuffer = *pucRegBuffer >> (8 - usNCoils);
            break;

            /* write current coil values with new values from the protocol stack. */
        case MB_REG_WRITE:
            while (iNReg > 1)
            {
                xMBUtilSetBits(&pucCoilBuf[iRegIndex++], iRegBitIndex, 8,
                        *pucRegBuffer++);
                iNReg--;
            }
            /* last coils */
            usNCoils = usNCoils % 8;
            /* xMBUtilSetBits has bug when ucNBits is zero */
            if (usNCoils != 0)
            {
                xMBUtilSetBits(&pucCoilBuf[iRegIndex++], iRegBitIndex, usNCoils,
                        *pucRegBuffer++);
            }
            break;
        }
    }
    else
    {
        eStatus = MB_ENOREG;
    }
    return eStatus;
}
```

- Callback function for discrete inputs:

The callback function handles the instructions associated with the discrete inputs.

```
eMBErrorCode eMBRegDiscreteCB( UCHAR * pucRegBuffer, USHORT usAddress, USHORT
usNDiscrete )
{
    eMBErrorCode    eStatus = MB_ENOERR;
    USHORT          iRegIndex , iRegBitIndex , iNReg;
    UCHAR *         pucDiscreteInputBuf;
    USHORT          DISCRETE_INPUT_START;
    USHORT          DISCRETE_INPUT_NDISCRETES;
    USHORT          usDiscreteInputStart;
    iNReg =  usNDiscrete / 8 + 1;

    pucDiscreteInputBuf = ucSDiscInBuf;
    DISCRETE_INPUT_START = S_DISCRETE_INPUT_START;
    DISCRETE_INPUT_NDISCRETES = S_DISCRETE_INPUT_NDISCRETES;
    usDiscreteInputStart = usSDiscInStart;

    /* it already plus one in modbus function method. */
    usAddress--;

    if ((usAddress >= DISCRETE_INPUT_START)
            && (usAddress + usNDiscrete    <= DISCRETE_INPUT_START +
DISCRETE_INPUT_NDISCRETES))
    {
        iRegIndex = (USHORT) (usAddress - usDiscreteInputStart) / 8;
        iRegBitIndex = (USHORT) (usAddress - usDiscreteInputStart) % 8;

        while (iNReg > 0)
        {
            *pucRegBuffer++ = xMBUtilGetBits(&pucDiscreteInputBuf[iRegIndex++],
                    iRegBitIndex, 8);
            iNReg--;
        }
        pucRegBuffer--;
        /* last discrete */
        usNDiscrete = usNDiscrete % 8;
        /* filling zero to high bit */
        *pucRegBuffer = *pucRegBuffer << (8 - usNDiscrete);
        *pucRegBuffer = *pucRegBuffer >> (8 - usNDiscrete);
    }
    else
    {
        eStatus = MB_ENOREG;
    }

    return eStatus;
}
```

- Callback function for holding registers:

The callback function handles the instructions associated with the holding registers.

```
eMBErrorCode eMBRegHoldingCB(UCHAR * pucRegBuffer, USHORT usAddress,
        USHORT usNRegs, eMBRegisterMode eMode)
{
    eMBErrorCode    eStatus = MB_ENOERR;
    USHORT          iRegIndex;
    USHORT *        pusRegHoldingBuf;
    USHORT          REG_HOLDING_START;
    USHORT          REG_HOLDING_NREGS;
    USHORT          usRegHoldStart;

    pusRegHoldingBuf = usSRegHoldBuf;
    REG_HOLDING_START = S_REG_HOLDING_START;
    REG_HOLDING_NREGS = S_REG_HOLDING_NREGS;
    usRegHoldStart = usSRegHoldStart;

    /* it already plus one in modbus function method. */
    usAddress--;

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

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

```
}
```

- Callback function for input register:

The callback function handles the instructions associated with the input register.

```
eMBErrorCode eMBRegInputCB(UCHAR * pucRegBuffer, USHORT usAddress, USHORT usNRegs )
{
    eMBErrorCode    eStatus = MB_ENOERR;
    USHORT          iRegIndex;
    USHORT *        pusRegInputBuf;
    USHORT          REG_INPUT_START;
    USHORT          REG_INPUT_NREGS;
    USHORT          usRegInStart;

    pusRegInputBuf = usSRegInBuf;
    REG_INPUT_START = S_REG_INPUT_START;
    REG_INPUT_NREGS = S_REG_INPUT_NREGS;
    usRegInStart = usSRegInStart;

    /* it already plus one in modbus function method. */
    usAddress--;

    if ((usAddress >= REG_INPUT_START)
            && (usAddress + usNRegs <= REG_INPUT_START + REG_INPUT_NREGS))
    {
        iRegIndex = usAddress - usRegInStart;
        while (usNRegs > 0)
        {
            *pucRegBuffer++ = (UCHAR) (pusRegInputBuf[iRegIndex] >> 8);
            *pucRegBuffer++ = (UCHAR) (pusRegInputBuf[iRegIndex] & 0xFF);
            iRegIndex++;
            usNRegs--;
        }
    }
    else
    {
        eStatus = MB_ENOREG;
    }

    return eStatus;
}
```

3. Software and Hardware Requirements

3.1 Software Requirements

- BSP version
  - M460 Series BSP CMSIS V3.00.001
- IDE version
  - Keil uVersion 5.37

3.2 Hardware Requirements

- Circuit components
  - NuMaker-M467HJ V1.0
- Pin Connect
  - Connect the UART0 TX (PB.13) pin to the PC UART RX pin, and connect M467 to the PC MAC port by an Ethernet cable.

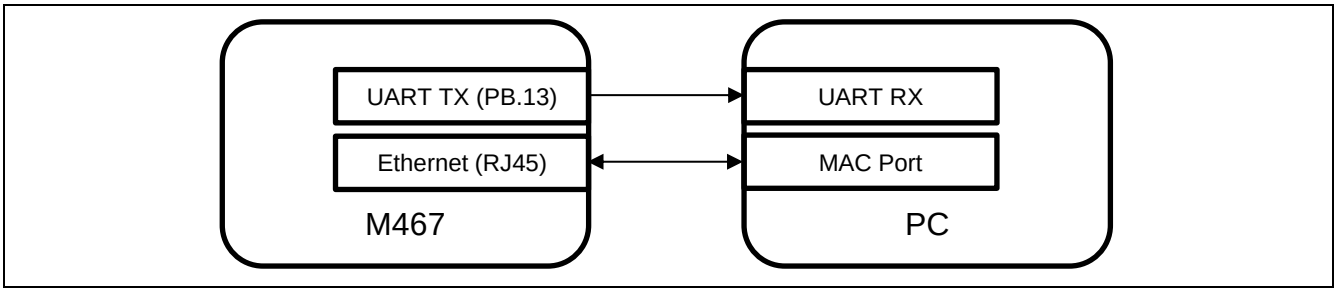

Figure 3-1 Pin Connect

4. Directory Information

The directory structure is shown below.

EC_M467_LwIP_Modbus_TCP_V1.00	
Library	Sample code header and source files
CMSIS	Cortex® Microcontroller Software Interface Standard (CMSIS) by Arm® Corp.
Device	CMSIS compliant device header file
StdDriver	All peripheral driver header and source files
SampleCode	
ExampleCode	Source file of example code
ThirdParty	
FreeRTOS	A real time operating system available for free download. Its official website is: http://www.freertos.org/
lwIP	A widely used open source TCP/IP stack designed for embedded systems. Its official website is: http://savannah.nongnu.org/projects/lwip/
modbus	FreeModbus Libary: A portable Modbus implementation for Modbus ASCII/RTU: https://www.embedded-experts.at/en/freemodbus/about/

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 *LwIP_Modbus_TCP.uvprojx*.
2. Enter Keil compile mode.
   - Build
   - Download
   - Start/Stop debug session
3. Enter debug mode.
   - Run

6. Revision History

Date	Revision	Description
2023.06.24	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.