AN0131
Application Note

Modbus on AT32 MCU

Introduction

This application note describes how to migrate or move FreeMODBUS protocol to AT32F43x microcontroller. Source code included in this user manual demonstrates how the Modbus program works in which the microcontroller as a Modbus slave is connected with a host computer via RS485 or RS232 so as to communicate with Modbus Poll debugging tool as a Modbus master.

Note: The codes included in this file are built around ARTERY V2.x.x BSP. Attention should be paid to the possible differences in use due to different BSP versions.

Applicable products:

Model	AT32F435xx
	AT32F437xx
	AT32F425xx

Contents

List of tables

List of figures

1 Overview

Modbus is an industrial-standard serial communications protocol. It is published by Modicon Company (now Schneider Electric) in 1979 for communication over programmable logic controllers.

Modbus is regarded as an industry-standard communications protocol and a popular form of connection between industrial electronic devices.

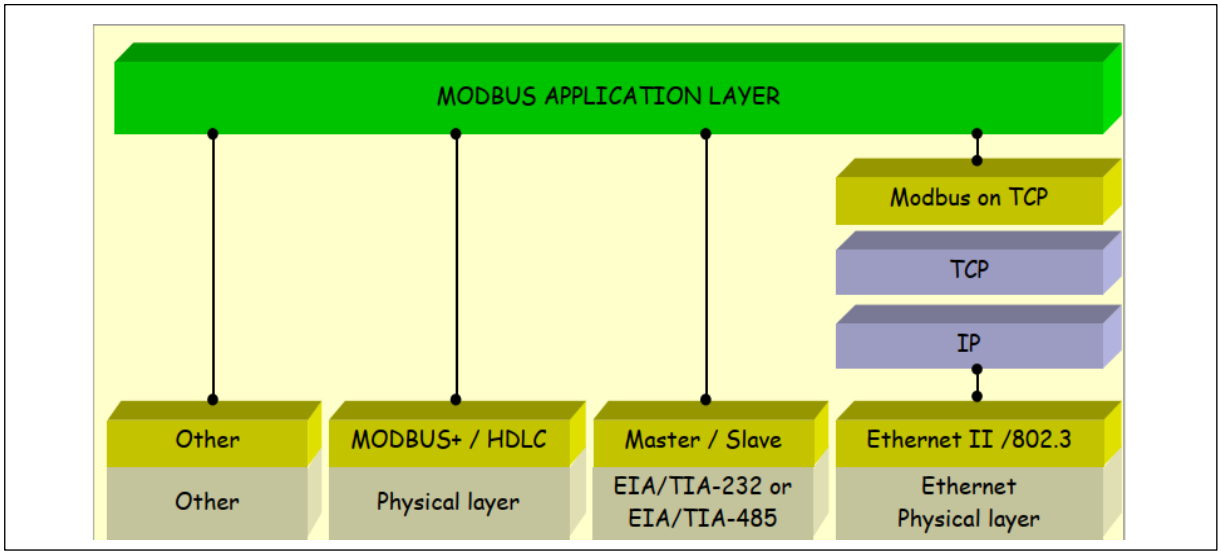
1.1 Modbus protocol

Modbus protocol uses master and slave mode for communications. A master takes initiative to query and operate a slave. Modbus Master refers to the protocols used by a master device, whereas Modbus Slave refers to the protocols used by a slave device. Typical master devices include industrial control computers and industrial controllers. Programmable logic controller (PLC) is used as a typical slave device.

MODBUS is an application layer messaging protocol, positioned at level 7 of the OSI model, which provides client/server communication between devices connected on different types of buses or networks.

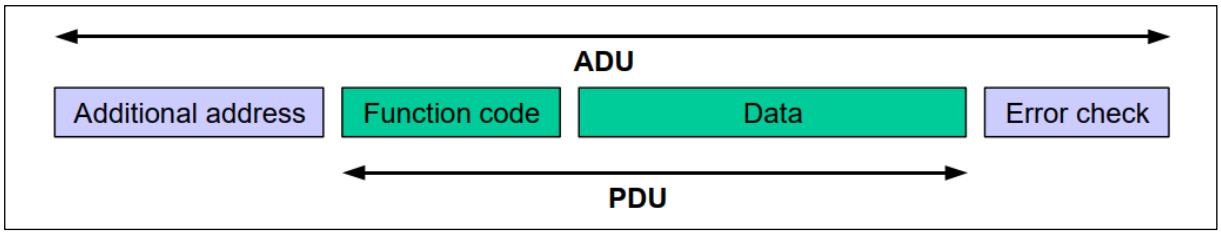
Physical interfaces for Modbus protocol can be serial interfaces (including RS232, RS485) and Ethernet.

Figure 1. Modbus communication stack



The MODBUS protocol defines a simple protocol data unit (PDU) independent of the underlying communication layers. The mapping of MODBUS protocol on specific buses or network can introduce some additional fields on the Application Data Unit (ADU).

Figure 2. General Modbus frame



When the server responds to the client, it uses the function code field to indicate either a normal response (error-free) or that some kind of error occurred (called an exception response). It is desirable to manage a time out in order not to indefinitely wait for an answer which will perhaps never arrive.

Figure 3. Modbus transaction (error free)

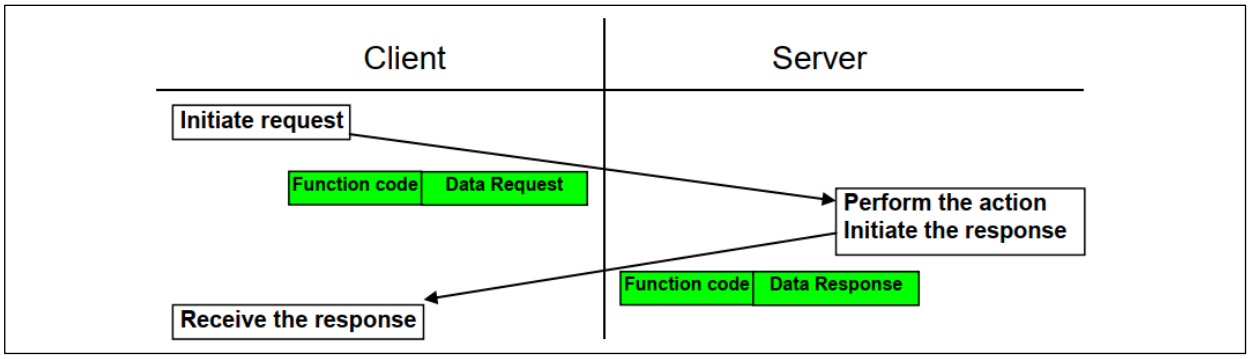
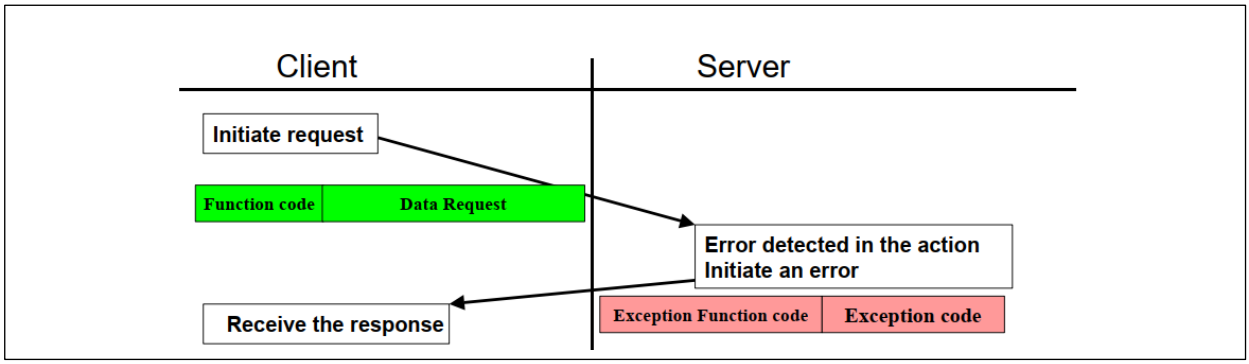


Figure 4. Modbus transaction (exception error)



MODBUS uses a “big-Endian” representation for addresses and data items. This means that when a numerical quantity larger than a single byte is transmitted, the most significant byte is sent first. So for example, if a 16-bit register value is 0x1234, then the 0x12 is sent first, followed by 0x34.

MODBUS bases its data model on a series of tables that have distinguishing characteristics. The four primary tables are:

Table 1. Modbus data model

Primary tables	Object type	Access type	Descriptions
Discrete input	Single bit	Read-only	This type of data can be provided by an I/O system.
Coils	Single bit	Read-Write	This type of data can be alterable by an application program.
Input registers	16bit word	Read-only	This type of data can be provided by an I/O system.
Holding registers	16bit word	Read-Write	This type of data can be alterable by an application program.

There are three categories of MODBUS Function codes. They are: public function codes, user-defined function codes and reserved function codes

Public function codes are well-defined function codes, guaranteed to be unique, validated by the MODBUS.org community, and have available conformance test.

Figure 5. Public function code definition

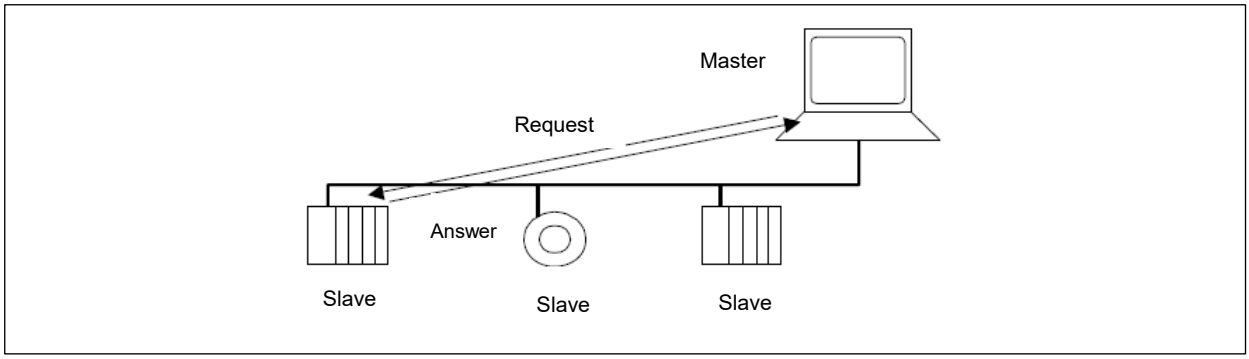
				Function codes		
				Code	Sub code	Hex
Data access	Bit access	Physical discrete inputs	Read discrete inputs	02		02
		Internal bits or physical coils	Read coils	01		01
			Write single coil	05		05
			Write multiple coils	15		0F
	16-bits access	Physical input registers	Read input register	04		04
		Internal registers or physical output registers	Read multiple register	03		03
			Write single register	06		06
			Write multiple register	16		10
			Read/write multiple registers	23		17
			Mask write register	22		16
	File record access		Read file record	20	6	14
			Write file record	21	6	15
	Package interface		Read device identification	43	14	2B

Modbus serial link protocol is a master/slave protocol. Only one master node is connected to a serial bus at one time, whereas one or several slave nodes (numbered up to 247) are linked to the same bus. Modbus communication is always initiated with a master node. Slave nodes never send data unless they receive requests from a master node. No communication takes place between slave nodes. A master node sends only one Modbus transaction at one time.

A master node sends a Modbus request to slave nodes in the following two modes:

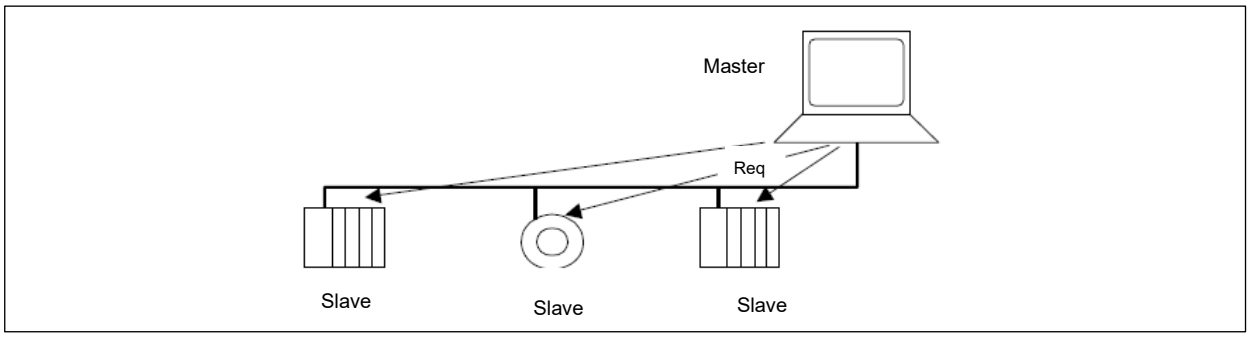
1. **Unicast mode**: A master node access a slave node at a given address. After receiving and handling a request from a master node, the slave node returns a message (known as response or answer) to the master node. Each of the slave nodes must have its own unique address ranging from 1 to 247 in order to be addressed accurately.

Figure 6. Unicast mode



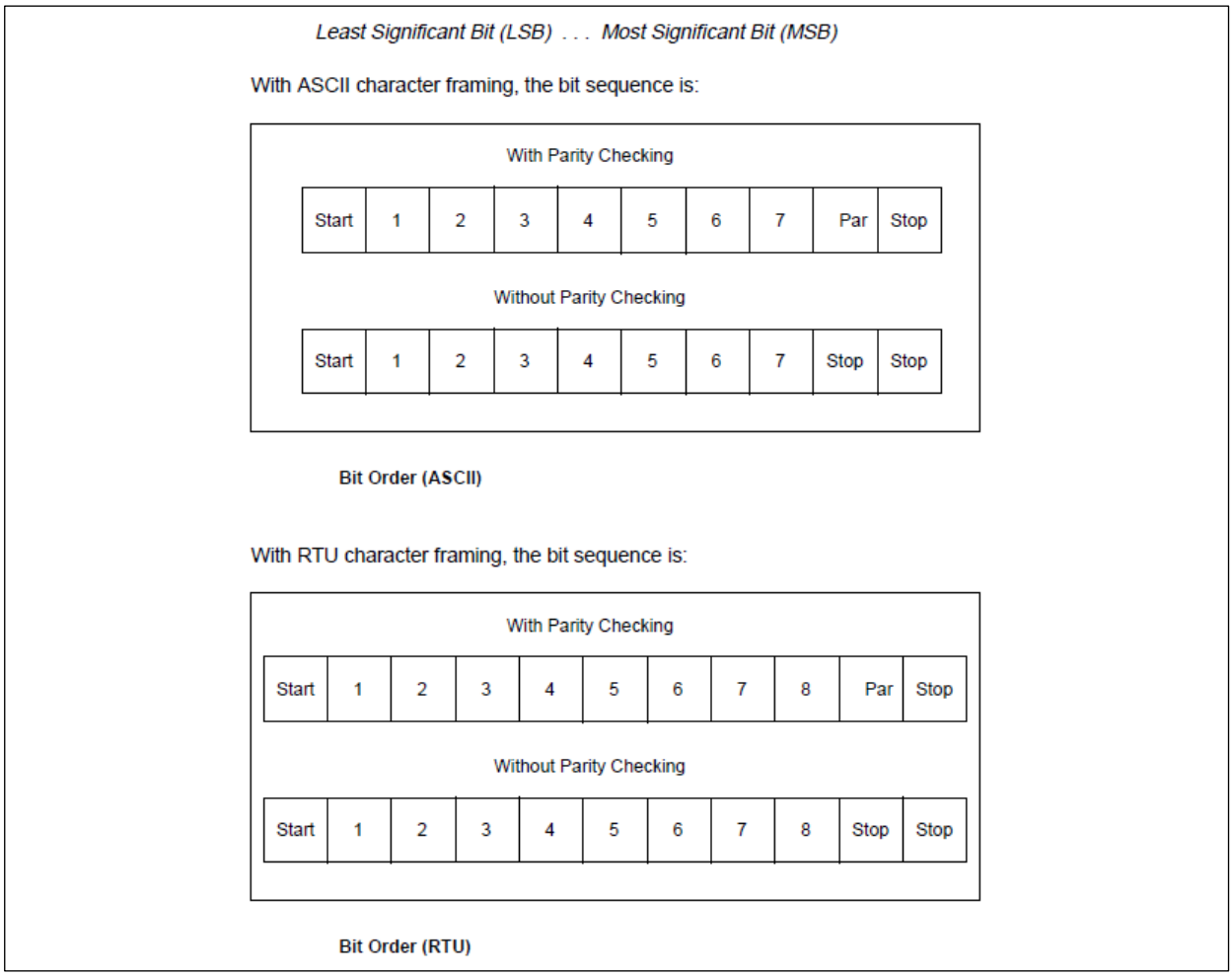
2. **Broadcast mode**: A master node sends a request to all the slave nodes. In this case, no response is returned/sent to the master node. Broadcast request is usually used for writing commands. All the devices must accept “write function” of broadcast mode. Address 0 is dedicated to a representation of broadcast data.

Figure 7. Broadcast mode



Modbus defines two serial transport modes: RTU (default) and ASCII.

Figure 8. Bit order in RTU and ASCII



All the devices on Modbus serial line shall have the same transport modes and serial port parameters. For more information about Modbus, please visit Modbus official website at https://modbus.org.

1.2 FreeModbus protocol stack

FreeMODBUS is an implementation of the popular Modbus communication protocol specially targeted for embedded systems. It supports RTU/ASCII modes and TCP protocol. FreeMODBUS is available under BSD license, meaning that it can be applied in business scenarios. At present, FreeMODBUS offers only one free protocol stack for Modbus slave nodes. This protocol is written with ANSI C and supports multiple variables.

The following sections will provide users with information on how to implement the major features of Modbus slave nodes on the AT32F435 microcontroller through FreeMODBUS protocol. The source codes based on AT32F43x_StdPeriph_Lib and FreeMODBUS are provided as well. Meanwhile, it also demonstrates how to quickly establish RS485-based Modbus slave nodes using AT32-Comm-EV Board and AT-START Board.

1.3 Modbus Poll debugging software

Modbus Poll is a Modbus master simulator. It supports Modbus RTU, ASCI and TCP/IP transport modes. It can be used to assist engineers in debugging Modbus slave devices, testing and simulating Modbus protocol communications. With multiple file interfaces/windows, Modbus Poll is able to monitor multiple Modbus slave devices and data fields simultaneously. Each interface or window offers such setting options as slave device ID, function code, address code, length and poll interval. Modbus Poll supports four primary tables of Modbus data models and multiple public function codes.

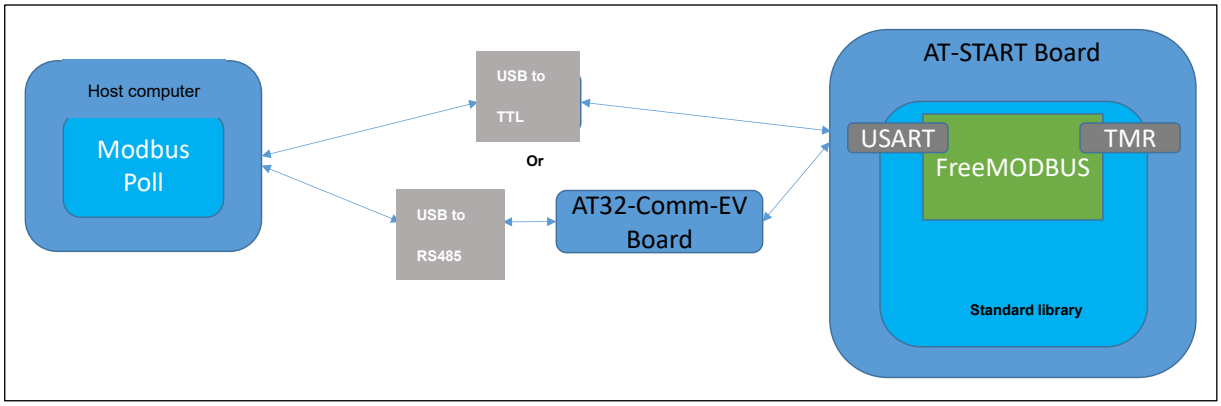
In the example case, we download and install Modbus Poll on PC as a Modbus master, which is connected to AT-START Board (as a Modbus slave) via USB-to-RS485 module so as to establish a complete Modbus communication network for testing.

2 AT32 hardware requirements

Hardware includes AT32-Comm-EV Board and AT-START Board.

The demo in this file uses such peripherals as USART and TMR. Users can also select RS232 or RS485 for connecting Modbus physical layer according to their needs.

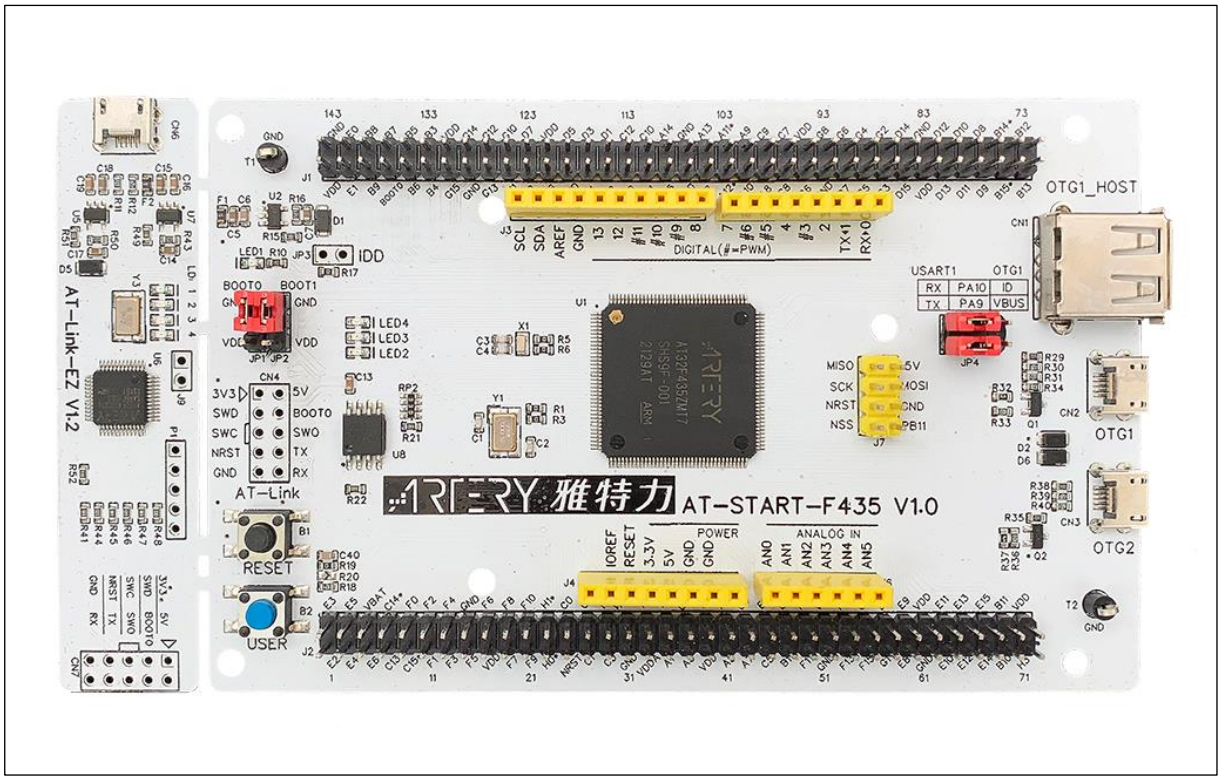
Figure 9. AT32 Modbus structure block diagram



- AT-START Board

The demo is based on AT-START-F435 board. It can provide RS232-based Modbus communications.

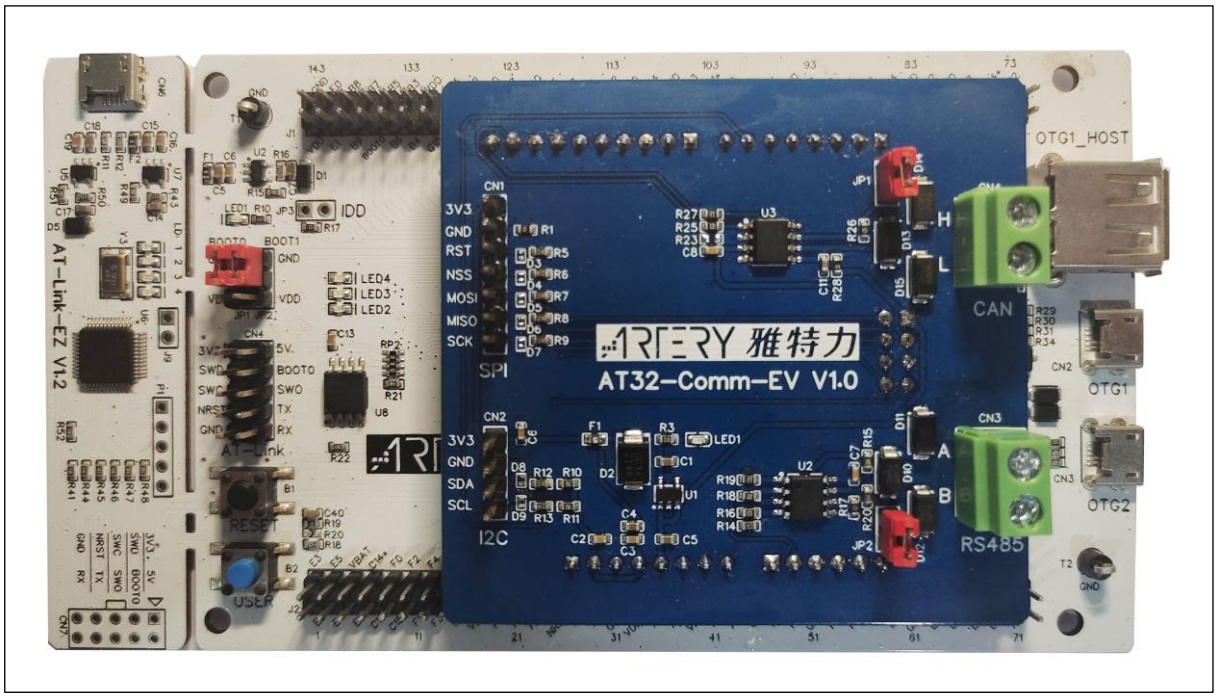
Figure 10. AT-START-F435 V1.0 evaluation board



- AT32-Comm-EV Board

This board supports RS485-based Modbus communications.

Figure 11. AT-START and AT32-Comm-EV



3 Move FreeMODBUS to AT32 MCU

3.1 Get started

Download and install the latest version of BSP&PACK files, install and configure according to user manuals.

This application note and example case are written with *BSP&PACK of the AT32F4xx_StdPeriph_Lib_V2.x.x.*

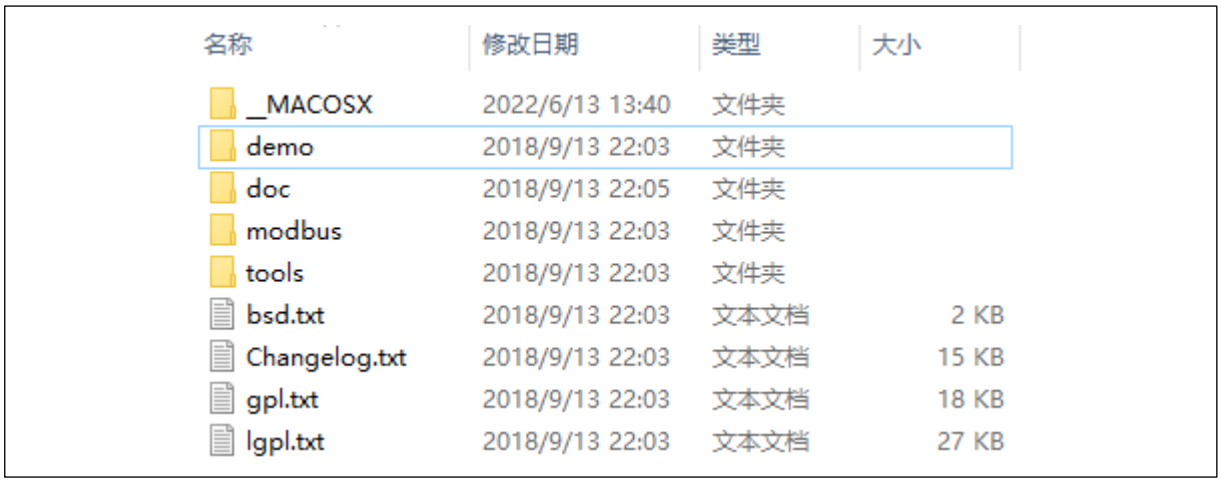
In the *at_start_f435* folder there is a *temple* project which can be used as a reference for users to make corresponding modifications. Change folder name and project name to freemodbus, and add FreeMODBUS source code.

3.2 Add FreeMODBUS source code

Download the latest version of FreeMODBUS source code from FreeMODBUS website or Github. After unzipping the package, you can see the following contents in it, as shown in Figure 12.

This application note and example case are based on freemodbus-v1.6.

Figure 12. FreeMODBUS source code files



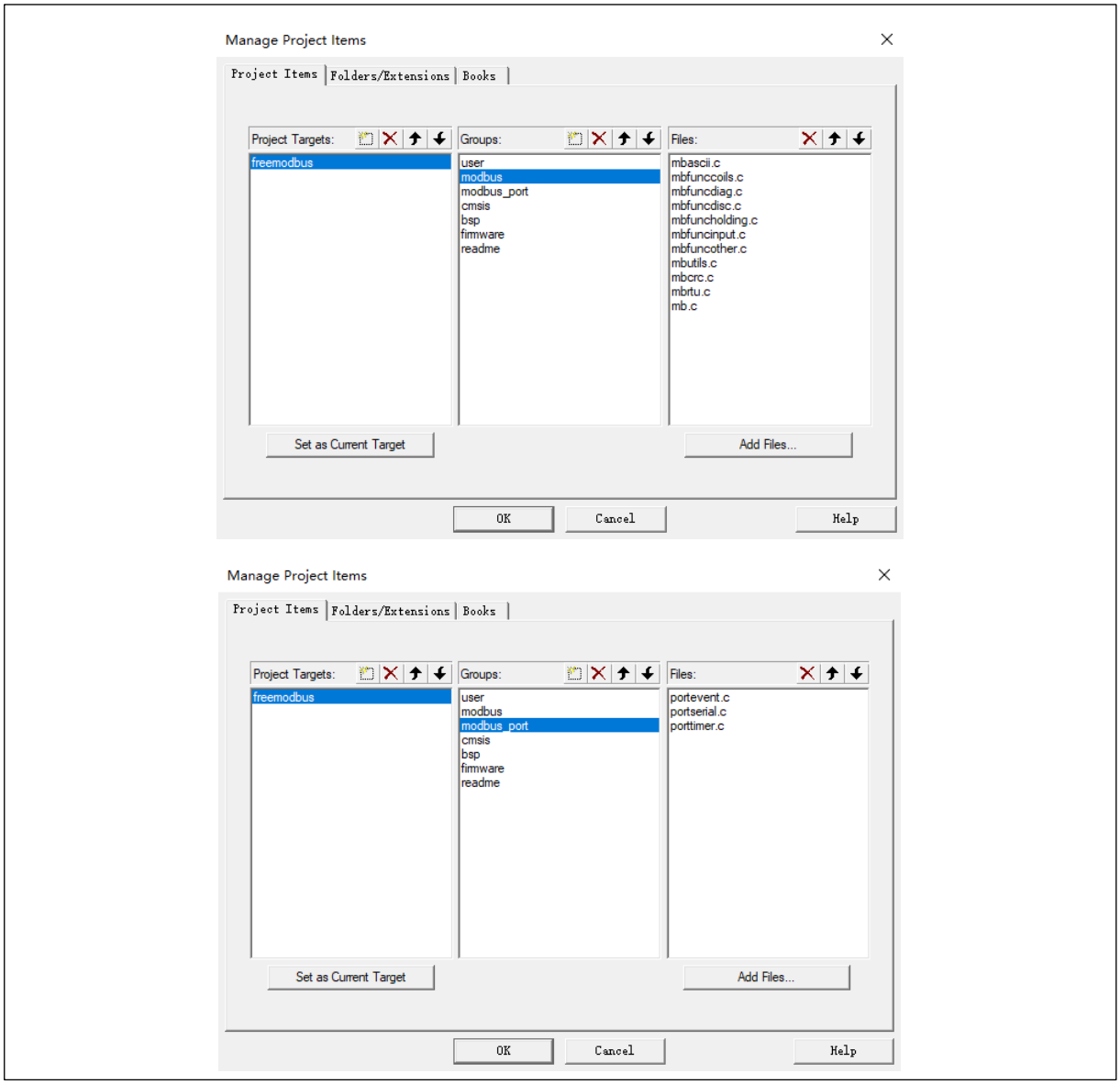
After unzipping, copy *modbus* folder and *demo\BARE\port* folder to the *freemodbus* (previously created), change *port* folder name to *modbus_port* as shown in Figure 13.

Figure 13. freemodbus project folder

Open project files, and follow two steps below to proceed. Users can also refer to the example case provided by ARTERY for more information.

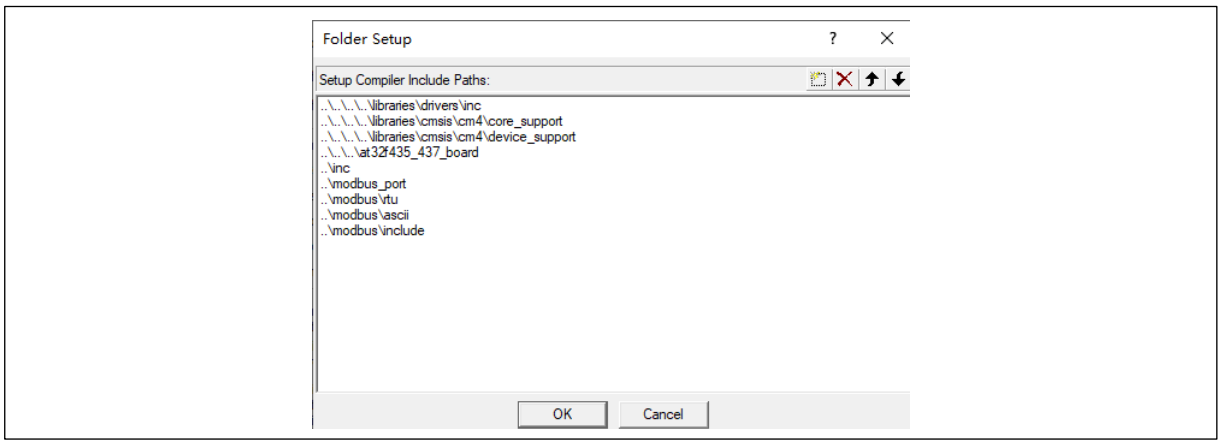
1. Add all ***.c*** files (excluding tcp) within *Modbus* and *modbus_port* folder to project

Figure 14. freemodbus project items



2. Add the path of the ***.h*** files corresponding to ***.c*** files into the Folder setup.

Figure 15. freemodbus project folder setup



3.3 How to change project code

1. **Change "*port.h*" file**. Add the header file "*at32f435_437.h"* to the "*port.h*" file. Supplement interrupts enable/disable macro definitions. Mask TRUE and FALSE definitions as they are already defined in BSP header file.
2. **Change "*portserial.c*" and "*porttimer.c*" files**. Add the underlying driver codes of USART and TMR peripherals to these files. Users can also modify according to their hardware environment or refer to ARTERY-provided example case.
3. **Special attention:** The DT field in the USRAT register contains data bit and check bit. Therefore when reading data received via USART, the FreeMODBUS source code would handle all DT values as data bits because of the differences of MCU from supplier to supplier. For this reason, it is necessary to change source code in the "*mbascii.c*" file. For more information, please refer to the demo.
4. **Create and add "*mbtask.c/.h*" file**. This file is used for creating a Modbus communication task (as a slave) to call API layer of the FreeMODBUS protocol, and build Modbus data models (four primary tables) so as to conduct communication testing with Modbus Poll (as a master).

"***mbtask.c/.h***" can be used for the following tasks:

- Read/write holding registers
- Read input registers
- Read/write coils
- Read discrete inputs

"***mbtask.h***" defines Modbus data models and parameters necessary for communication:

Table 2. Modbus configuration parameters

Parameter	Description
MB_SLAVE_ADDRESS	Set slave address
MB_BAUDRATE	Set communication baud rate
REG_INPUT_START	Input register start address
REG_INPUT_NREGS	Number of Input register s
REG_HOLDING_START	Holding register start address
REG_HOLDING_NREGS	Number of holding registers
REG_COILS_START	Coil start address
REG_COILS_SIZE	Coil size
REG_DISCRETE_START	Discrete input start address
REG_DISCRETE_SIZE	Discrete input size

3.4 How to implement device functions

1. Write the "*void modbus_task(void)*" function in the "*mbtask.c*" file to call API layer of Modbus protocol so as to conduct Modbus slave tasks.

```
void modbus_task(void)
{
  eMBErrorCode       eStatus;

  eStatus = eMBInit(MB_RTU, MB_SLAVE_ADDRESS, 0, MB_BAUDRATE, MB_PAR_NONE);
  if(MB_ENOERR == eStatus)
  {
    printf("modbus init ok\r\n");
    eStatus = eMBEnable();
    if(MB_ENOERR == eStatus)
    {
      printf("modbus enable ok\r\n");
    }
    else
    {
      printf("modbus enable fail, error code: %u\r\n", eStatus);
    }
  }
  else
  {
    printf("modbus init fail, error code: %u\r\n", eStatus);
  }
  if(MB_ENOERR != eStatus)
  {
    printf("exit modbus task.\r\n");
    return;
  }
  printf("start modbus pooling..\r\n");
  for(;;){
    eMBPoll();
  }
}
```

2. In “*main.c”* file, use the “*int main(void)*” to call the “*modbus_task()*” function.

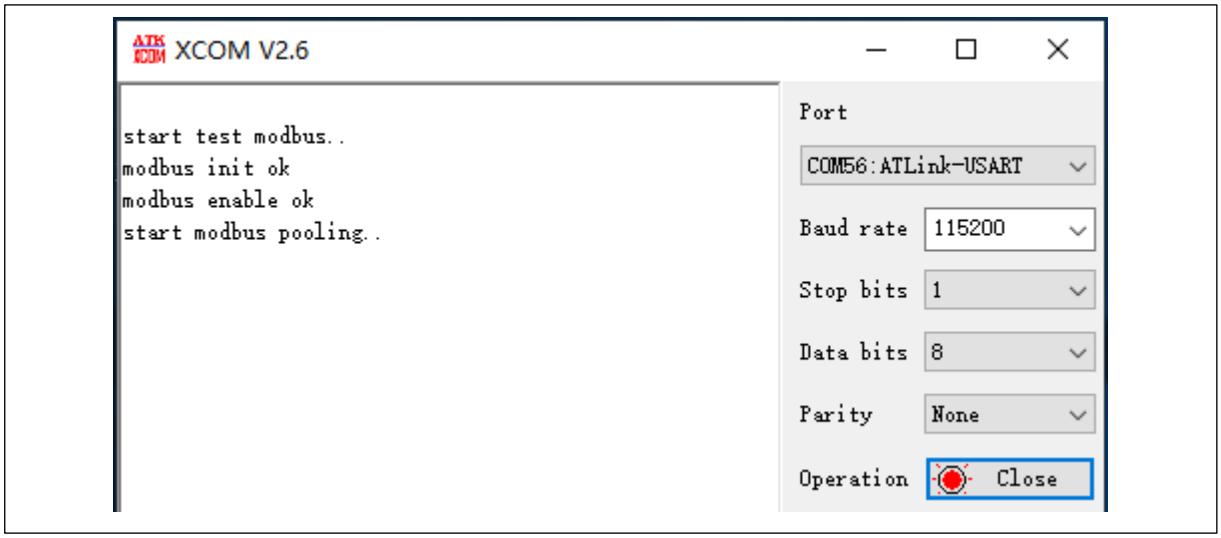
```
int main(void)
{
  system_clock_config();
  at32_board_init();
  uart_print_init(115200);
  printf("\r\nstart test modbus..\r\n");

  modbus_task();
  while(1);
}
```

4 Device testing

Source code migration is complete until now. Compile and download it, open serial interfaces connected to AT-Link, the following information will be print out on the window.

Figure 16. Serial interface print info

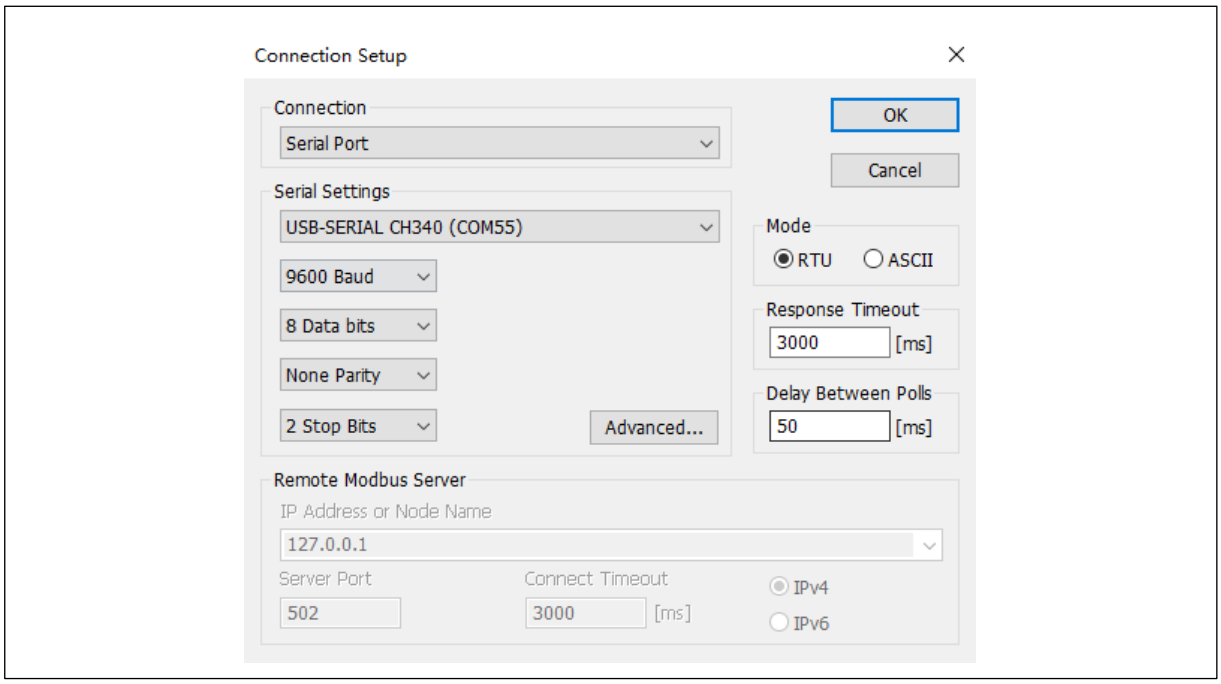


This indicates that a slave device is running normally.

At this point, connect the slave to a host computer, open Modbus Poll software, and use it as a master for unicast communication, which is done by sending a request and receiving a reply.

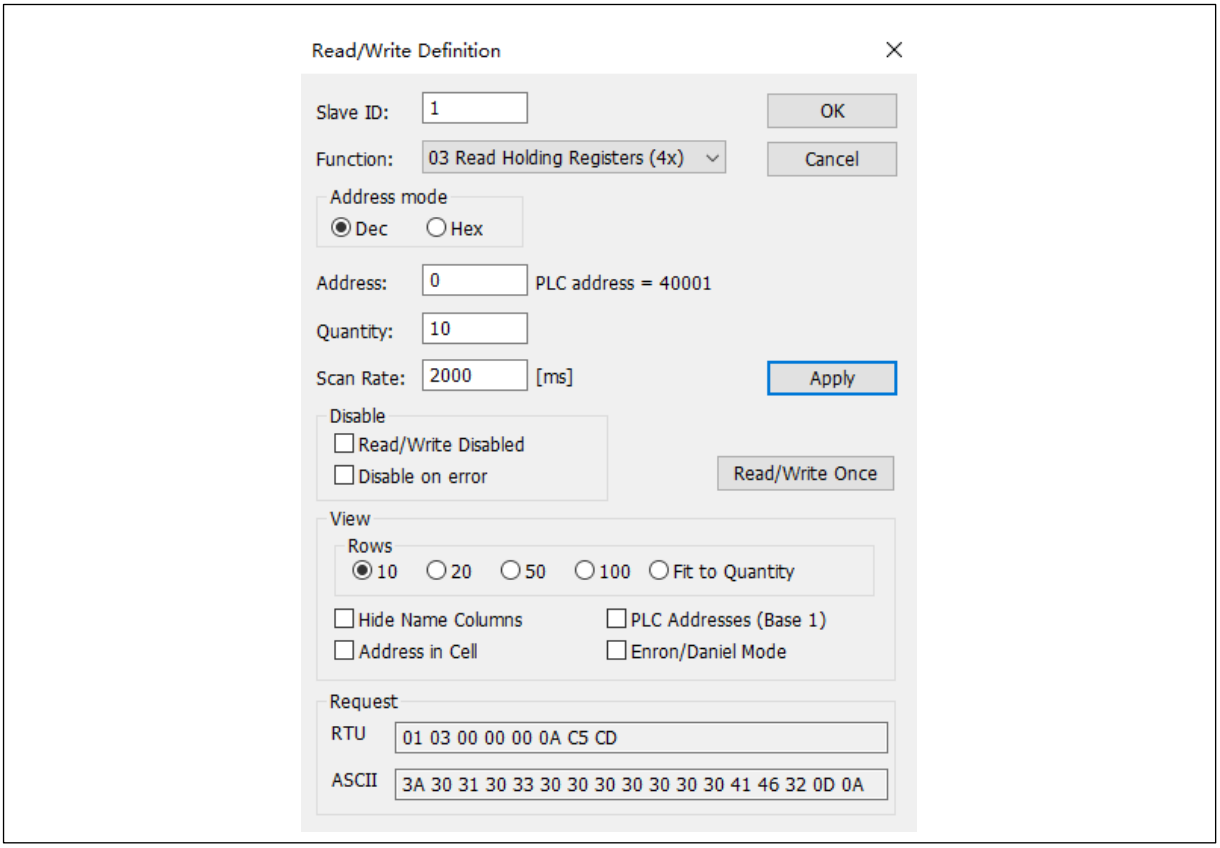
1. Open Modbus Poll connection setup, select RTU Mode (same as a slave), and configure serial port parameters (they must be the same as those of slave)

Figure 17. Modbus Poll connection setup



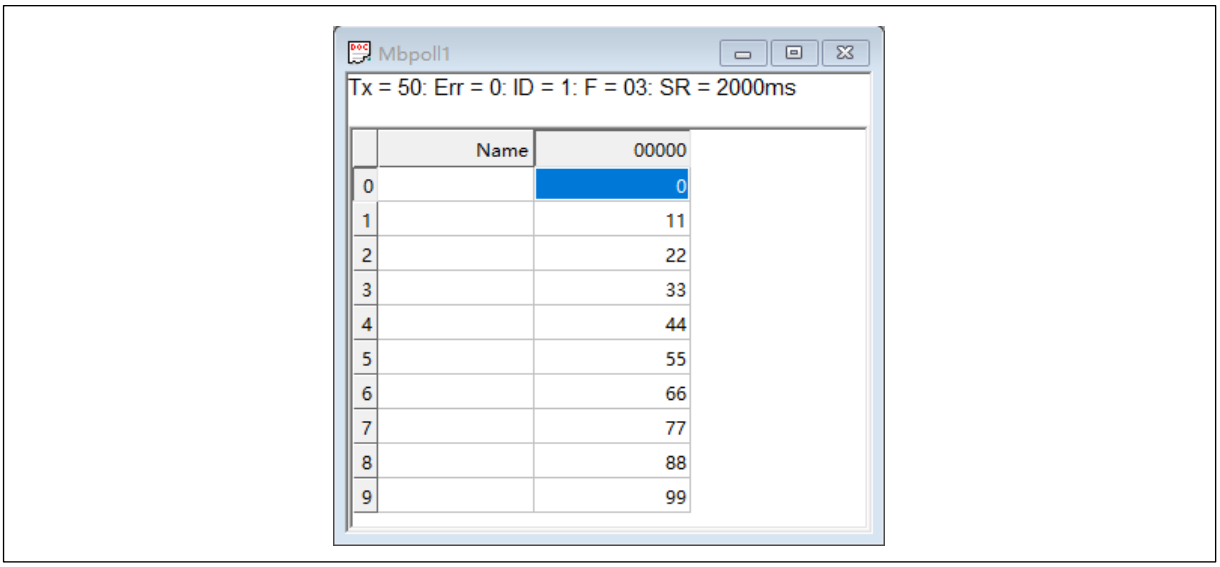
2. Define Modbus Poll read/write commands. Figure 18 shows an example with a 03 function code (read holding registers). Users can also test other function codes in the same way.

Figure 18. Modbus Poll read/write definition



3. In Modbus Poll file interface/window, you can see the values read from holding registers. These data are consistent with the ones of holding registers when they were initialized in a slave program. This means that test is successful.

Figure 19. Modbus Poll file interface



5 Revision history

Table 3. Document revision history

Date	Revision	Changes
2022.06.13	2.0.0	Initial release

IMPORTANT NOTICE – PLEASE READ CAREFULLY

Purchasers are solely responsible for the selection and use of ARTERY's products and services, and ARTERY assumes no liability whatsoever relating to the choice, selection or use of the ARTERY products and services described herein.

No license, express or implied, to any intellectual property rights is granted under this document. If any part of this document deals with any third party products or services, it shall not be deemed a license grant by ARTERY for the use of such third party products or services, or any intellectual property contained therein, or considered as a warranty regarding the use in any manner whatsoever of such third party products or services or any intellectual property contained therein.

Unless otherwise specified in ARTERY's terms and conditions of sale, ARTERY provides no warranties, express or implied, regarding the use and/or sale of ARTERY products, including but not limited to any implied warranties of merchantability, fitness for a particular purpose (and their equivalents under the laws of any jurisdiction), or infringement of any patent, copyright or other intellectual property right.

Purchasers hereby agrees that ARTERY's products are not designed or authorized for use in: (A) any application with special requirements of safety such as life support and active implantable device, or system with functional safety requirements; (B) any air craft application; (C) any automotive application or environment; (D) any space application or environment, and/or (E) any weapon application. Purchasers' unauthorized use of them in the aforementioned applications, even if with a written notice, is solely at purchasers' risk, and is solely responsible for meeting all legal and regulatory requirement in such use.

Resale of ARTERY products with provisions different from the statements and/or technical features stated in this document shall immediately void any warranty grant by ARTERY for ARTERY products or services described herein and shall not create or expand in any manner whatsoever, any liability of ARTERY.