Research on gateway between EtherCAT and Modbus RTU

zhuzhi FAN

Supervisor: Yanju ZHU,Minh Tu Pham

A report of engineer internship

in the

Department of Mechanical Engineering

February 1,2018

Preface

Currently I am completing a double degree (Bachelor's degree in NPU, engineer's degree in INSA de Lyon). I'm in my 5rd year and therefore a 6-month internship should be conducted. The reason to choose my internship on a theme of the CNC, Computerized Numerical Control Machine, is: that machine used to make components of other machines automatically has always fascinated me since high school. It's therefore amazing if improvement of one CNC system can be acted in the internship with my knowledge. In addition to that, the subject of this internship concerns principally the EtherCAT, Ethernet for Control Automation Technology. That technology is totally new to me, it's interesting to learn and challenge something completely new.

As my internship of designing that gateway is independent of works of my colleagues, to better handover my design and make it easier for others to continue the research: all key data structures and interfaces of all important functions were described in detail in this report. This report is therefore more like an instruction manual for users and for other developers than a report of my internship. In that reason, it's quite easy to realize that gateway after reading this report.

For this opportunity, I thank:

Yanju ZHU, who is the director of the digital control software development department and my internship instructor. I thank her for giving me the opportunity to follow my internship in Beijing Jingdiao. She had the kindness to accept me and guide me through my internship with advice, feedback and tips despite his busy schedule.

I really enjoyed my stay in Jingdiao. It's a great experience and I want to thank everyone for that.

Finally, I thank my School Internship Instructor, Minh Tu Pham. He helped and coached me during my internship by giving me feedback and tips on my school staffs and helping me arrange the school schedule.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction to the internship

For a lot of advantages of the fieldbus[7], in the next generation of CNC controlling system, Beijing Jingdiao wants to use it to take place of CANopen, which has been used for longtime. However, EtherCAT is so advanced that it has not been widely supported by a lot of sensors, they cannot consequently be directly connected to new generation of CNC controlling system, and hence a gateway needed as a temporary solution. Considering that the protocol of Modbus RTU is commonly supported by most devices that usually used in the CNC system, we propose to design a gateway between Modbus RTU and EtherCAT. With that gateway, we can connect all devices supporting Modbus RTU to the new system directly. This is our solution to advance the newly designed CNC system before the use of EtherCAT is well generalized.

1.2 Introduction to the company

Beijing Jingdiao Group was founded in 1994, is one of the national torch plans and key high-tech enterprises[1], integrates with scientific research, production, sales and service, and has always been committed to providing customers with the entire solutions including micro milling, drilling and grinding process. Its business scope has covered CNC machine, CNC system, CAD/CAM software, high-speed motorized spindle, high-precision rotating function unit and related fields. Main product is CNC Machine Tool which is widely used in more than 30 fields including 3C products, precision mold & die, woodcarving, medical equipment. For twenty years, with strong R&D strength and practical experience, Beijing Jingdiao has realized CNC Machine Tool' machining requirements with high-speed, high-precision and high-reliability. The cumulative sets of Jingdiao CNC Machine Tool sold has reached 70000, Beijing Jingdiao can produce more than ten thousand medium-size CNC machine tools annually and the annual output value is over 4 billion RMB, nearly 650 million

euro.

1.3 Introduction of EtherCAT

EtherCAT (Ethernet for Control Automation Technology) is a real-time Industrial Ethernet technology originally developed by Beckhoff Automation. The EtherCAT protocol which is disclosed in the IEC standard IEC61158 is suitable for hard and soft real-time requirements in automation technology, in test and measurement and many other applications.

The main focus during the development of EtherCAT was on short cycle times (100 µs), low jitter for accurate synchronization (1 µs) and low hardware costs.

1.3.1 Functional Principle

The EtherCAT master sends a telegram that passes through each node. Each EtherCAT slave device reads the data addressed to it on the fly, *figure*1.1, and inserts its data in the frame as the frame is moving downstream. The frame is delayed only by hardware propagation delay times. The last node in a segment (or branch) detects an open port and sends the message back to the master using Ethernet technology's full duplex feature.

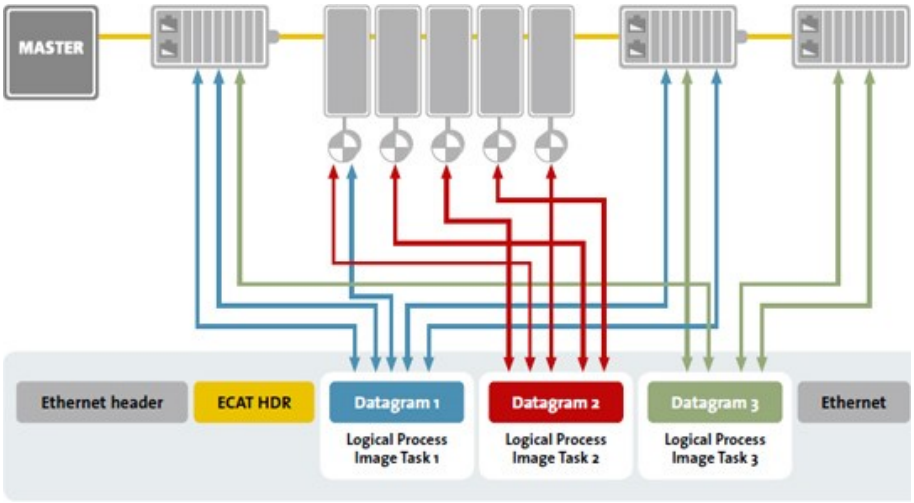

Figure 1.1: *Inserting process data on the fly*

The telegram's maximum effective data rate increases to over 90 %, and due to the utilization of the full duplex feature, the theoretical effective data rate is even higher than 100 Mbit/s (> 90 % of two times 100 Mbit/s).

The EtherCAT master is the only node within a segment allowed to actively send an EtherCAT frame; all other nodes merely forward frames downstream. This concept prevents unpredictable delays and guarantees real-time capabilities.

The master uses a standard Ethernet Media Access Controller (MAC) without an additional communication processor. This allows a master to be implemented on any hardware platform with an available Ethernet port, regardless of which real-time operating system or application software is used. EtherCAT Slave devices use an EtherCAT Slave Controller (ESC) to process frames on the fly and entirely in hardware, making network performance predictable and independent of the individual slave device implementation.

1.3.2 The EtherCAT Protocol

EtherCAT embeds its payload in a standard Ethernet frame, *figure*1.2, The frame is identified with the Identifier (0x88A4) in the EtherType field. Since the EtherCAT protocol is optimized for short cyclic process data, the use of protocol stacks, such as TCP/IP or UDP/IP, can be eliminated[2].

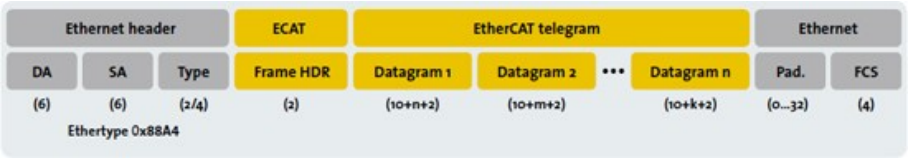

Figure 1.2: *EtherCAT in a standard Ethernet frame (according to IEEE 802.3)*

During startup, the master device configures and maps the process data on the slave devices. Different amounts of data can be exchanged with each slave, from one bit to a few bytes, or even up to kilobytes of data.

The EtherCAT frame contains one or more data-grams. The data-gram header indicates what type of access the master device would like to execute:

- Read, write, read-write
- Access to a specific slave device through direct addressing, or access to multiple slave devices through logical addressing (implicit addressing)

Logical addressing is used for the cyclical exchange of process data. Each Datagram addresses a specific part of the process image in the EtherCAT segment, for which 4GBytes of address space is available. During network startup, each slave device is assigned one or more addresses in this global address space. If multiple slave devices are assigned addresses in the same area, they can all be addressed with a single Datagram. Since the Datagrams completely contain all the data access related information, the master device can decide when and which data to access. For example, the master device can use short cycle times to refresh data on the drives, while using a longer cycle time to sample the I/O; a fixed process data structure is not necessary. This also relieves the master device in comparison to in conventional fieldbus systems, in which the data from each node had to be read individually, sorted with the help of the process controller, and copied into memory.

With EtherCAT, the master device only needs to fill a single EtherCAT frame with new output data, and send the frame via automatic Direct Memory Access (DMA) to the MAC controller. When a frame with new input data is received via the MAC controller, the master device can copy the frame again via DMA into the computer's memory all without the CPU having to actively copy any data. In addition to cyclical data, further Data-grams can be used for asynchronous or event driven communication.

In addition to logical addressing, the master device can also address a slave device via its position in the network. This method is used during network boot up to determine the network topology and compare it to the planned topology.

After checking the network configuration, the master device can assign each node a configured node address and communicate with the node via this fixed address. This enables targeted access to devices, even when the network topology is changed during operation, for example with Hot Connect Groups. There are two approaches for slave-to-slave communication. A slave device can send data directly to another slave device that is connected further downstream in the network. Since EtherCAT frames can only be processed going forward, this type of direct communication depends on the network's topology, and is particularly suitable for slave-to-slave communication in a constant machine design (e.g. in printing or packaging machines). In contrast, freely configurable slave-to-slave communication runs through the master device, and requires two bus cycles (not necessarily two control cycles). Thanks to EtherCAT's excellent performance, this type of slave-to-slave communication is still faster than with other communication technologies.

1.3.3 Why use EtherCAT?

The unique way that EtherCAT works makes it the clear engineer's choice. Additionally, the following features are particularly advantageous for many applications[2].

Exceptional performance

EtherCAT is by and large the fastest Industrial Ethernet technology, but it also synchronizes with nanosecond accuracy. This is a huge benefit for all applications in which the target system is controlled or measured via the bus system. The rapid reaction times work to reduce the wait times during the transitions between process steps, which significantly improves application efficiency. Lastly, the EtherCAT system architecture typically reduces the load on the CPU by 25 30 % in comparison to other bus systems (given the same cycle time). When optimally applied, EtherCAT's performance leads to improved accuracy, greater throughput, and thus to lowered costs.

Flexible topology

In EtherCAT applications, the machine structure determines the network topology, not the other way around. In conventional Industrial Ethernet systems, there are limitations on how many switches and hubs can be cascaded, which thus limits the overall network topology. Since EtherCAT does not need hubs or switches, there are no such limitations. In short, EtherCAT is virtually limitless when it comes to network topology. Line, tree, star typologies and any combinations thereof are possible with a nearly unlimited number of nodes. Thanks to automatic link detection, nodes and network segments can be disconnected during operation and then reconnected even somewhere else, if the master supports this feature. Line topology is extended to a ring topology for the sake of cable redundancy. All the master device needs for this redundancy is a second Ethernet port, and the slave devices already support the cable redundancy anyhow. This makes switching out devices during machine operation possible.

It's simple and robust

Configuration, diagnostics, and maintenance are all factors that contribute to system costs. The Ethernet fieldbus makes all of these tasks significantly easier: EtherCAT can be set to automatically assign addresses, which eliminates the need for manual configuration. A low bus load and peer-to-peer physics improve electromagnetic noise immunity. The network reliably detects potential disturbances down to their exact location, which drastically reduces the time needed for troubleshooting. During startup, the network compares the planned and actual layouts to detect any discrepancies. EtherCAT performance also helps during system configuration by eliminating the need for network tuning. Thanks to the large bandwidth, there is capacity to transmit additional TCP/IP together with the control data. However, since EtherCAT itself is not based on TCP/IP, there is no need to administer MAC-addresses or IP-addresses or to have IT experts configure switches and routers.

Integrated Safety

Functional safety as an integrated part of the network architecture? Not a problem with Functional Safety over EtherCAT (FSoE). FSoE is proven in use through TV certified devices that have been on the market since 2005. The protocol fulfills the requirements for SIL 3 systems and is suitable for both centralized and decentralized control systems. Thanks to the Black-Channel approach and the particularly lean Safety-Container, FSoE can also be used in other bus systems. This integrated approach and the lean protocol help keep system costs down. Additionally, a non-safety critical controller can also receive and process safety data.

Affordability

EtherCAT delivers the features of Industrial Ethernet at a price similar or even below that of a classic fieldbus system. The only hardware required by the master device is an Ethernet port no expensive interface cards or co-processors are necessary. EtherCAT slave controllers are available from various manufacturers in different formats: as an ASIC, based on FPGA, or also as an option for standard microprocessor series. Since these inexpensive controllers shoulder all the time-critical tasks, EtherCAT itself does not place any performance requirements on the CPU of slave devices, which keeps device costs down. Since EtherCAT does not require switches or other active infrastructure components, the costs for these components and their installation, configuration, and maintenance are also eliminated.

1.4 Introduction of Modbus

Modbus is a serial communications protocol originally published by Modicon (now Schneider Electric) in 1979 for use with its programmable logic controllers (PLCs). Modbus has become a de facto standard communication protocol and is now a commonly available means of connecting industrial electronic devices. The main reasons for the use of Modbus in the industrial environment are[3]:

- Developed with industrial applications in mind
- Openly published and royalty-free
- Easy to deploy and maintain
- Moves raw bits or words without placing many restrictions on vendors

Modbus enables communication among many devices connected to the same network, for example, a system that measures temperature and humidity and communicates the results to a computer. Modbus is often used to connect a supervisory computer with a remote terminal unit (RTU) in supervisory control and data acquisition (SCADA) systems. Many of the data types are named from its use in driving relays: a single-bit physical output is called a coil, and a single-bit physical input is called a discrete input or a contact.

The development and update of Modbus protocols has been managed by the Modbus Organization since April 2004, when Schneider Electric transferred rights to that organization. The Modbus Organization is an association of users and suppliers of Modbus-compliant devices that advocates for the continued use of the technology.

Nowadays, there are numbers of versions of Modbus protocol are used, containing Modbus RTU, Modbus ASCII, Modbus Plus and Modbus TCP. In this internship, we only focus in the Modbus RTU, which is the one the most often used between the two versions based on RS485, Modbus RTU and Modbus ASCII.

1.4.1 The protocol of Modbus RTU

Modbus object types

The following is a table of object types provided by a Modbus slave device to Modbus master device,*table*1.1:

Object type	Access	Size
Coil	Read-write	8 bits
Discrete input	Read-only	8 bits
Input register	Read-only	16 bits
Holding register	Read-write	16 bits

Table 1.1: *Table of object types of Modbus*

1.4.2 The frame format of Modbus RTU

Modbus object types

The following is a table of object types provided by a Modbus slave device to Modbus master device,*table*1.2:

Name	Length(bits)	Function
Start	28	At least 3.5-character times of silence (mark condition)
Address	8	Station address
Function	8	Indicates the function code
Data	N*8	Data +length will be filled depending on the message type
CRC	16	Cyclic redundancy check
End	28	At least 3.5-character times of silence (mark condition)

Table 1.2: *Table of the frame format of Modbus RTU*

The Modbus has strict requirement for the time of silence between each character sent, if the gap between two characters is longer than 1.5-character times, the device who receives the frame will consider it as a wrong signal. And if it's bigger than 3.5-character times, the device will consider it's the beginning of a new frame.

1.4.3 The function code of Modbus RTU

In our internship, those types of functions used are limited to 1(Read Coils) 2(Read Discrete Inputs), 3(Read Multiple Holding Registers), 4(Read Input Registers) 5(Write Single Coil), 6(Write Single Holding Register), 15(Write Multiple Coils) and 16(Write Multiple Holding

Function type			Function name	Function Code
Data Access	8-bit access	Physical Discrete Inputs	Read Discrete Inputs	22
		Internal Bits or Physical Coils	Read Coils	1
			Write Single Coil	5
			Write Multiple Coils	15
	16-bit access	Physical Input Registers	Read Input Registers	4
		Internal Registers or Physical Output Registers	Read Multiple Holding Registers	3
			Write Single Holding Register	6
			Write Multiple Holding Registers	16
			Read/Write Multiple Registers	23
			Mask Write Register	22
			Read FIFO Queue	24
	File Record Access		Read File Record	20
			Write File Record	21

Table 1.3: *Table of the function code of Modbus RTU*

Registers), which are the most basic functions to realize the communication and are that all Modbus devices support, *table*1.3,.

Chapter 2

A brief introduction of the gateway

I have realized two versions of gateway, one is based in the micro-controller, Freescale MK22FX512VLL12 and the chip of EtherCAT slave station, LAN9252I/ML, *figure*2.1. Another one is based in Infineon XMC4300, *figure*2.2 which contains the core of EtherCAT slave station inside it. For the first one, it's not a standard slave station, so we need to code every part for the EtherCAT slave station by ourselves, including coding for the data link layer. For the second one, it's a standard EtherCAT slave station developed by Infineon. With the latter, what we developers need to do is to focus on the application layer. Code of data link layer can be generated automatically, using the EtherCAT Slave Station Code Tool, SSC Tool, developed by Beckhoff. The main difference between the two final versions is there Object Dictionary (OD). To better introduce the detail of protocol of the gateway, I will base the report on the first version, whose data link layer are coded by myself.

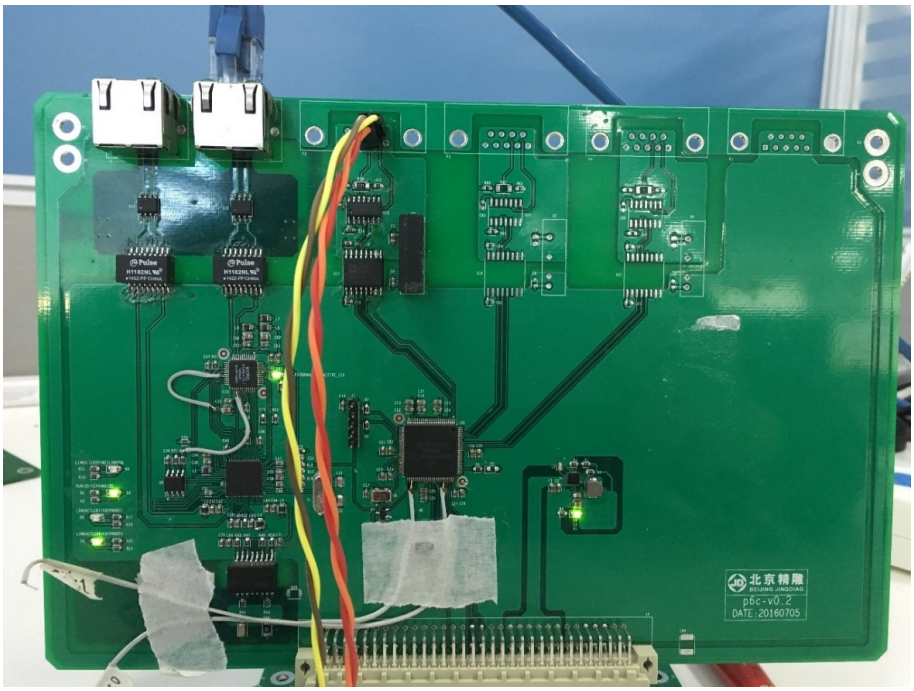

Figure 2.1: *Gateway based in Freescale MK22FX512VLL12 and LAN9252I/ML*

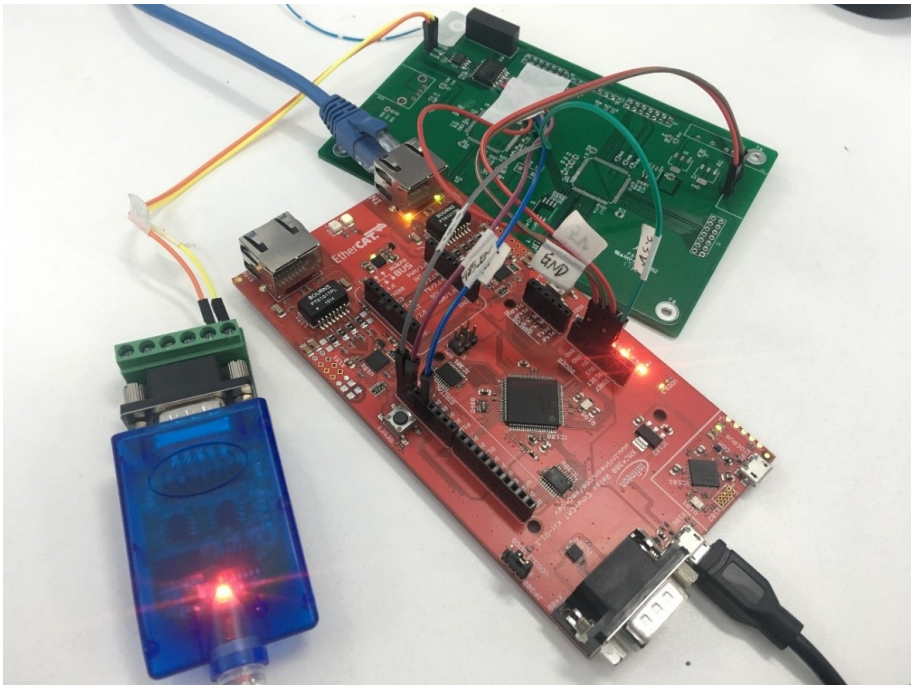

Figure 2.2: *Gateway based in Infineon XMC4300*

We use the gateway to connect the EtherCAT master (all the masters below refer to the

EtherCAT master, the slaves all refer to the EtherCAT slaves) and the Modbus slave devices (all slave devices below refer to Modbus slave devices,all mater devices below refer to Modbus master devices). So, the gateway is used as a **Modbus master device** and as a **EtherCAT slave** to realize the transforming between Modbus protocol and EtherCAT protocol.

In this version, the maximum scale of data input and output as 120 bytes. The data direction is defined from the Modbus side of view:

- Data from the gateway to the slave device defined as output data
- Data from the device to the gateway defined as input data

The default quantity of the slave devices is pre-set as 4. If quantity of devices is less than or equal to 4, there is no need to modify the code.The only exertion rest is to set the quantity and some other configuration by the EtherCAT master par SDO (Service Data Objects)[5]. The way to configure the gateway will be shown in section 3.1. When the quantity is more than 4, we need to change the relevant macro definition in the gateway program and add the corresponding OD (object dictionary). For details, see section 5, increase the maximum number of slaves supported by the gateway.

Chapter 3

The description and demonstration of process of using for gateway

3.1 Configure slave device and serial communication

3.1.1 Configure slave devices

What to configure includes the OD: 2000h, 2001h-200xh, x equal to the quantity of slave device connected to the gateway (The maximum number of gateways currently supported is 4), *figure*3.1.

2000:0	ModbusConfig	RO	> 1 <
2000:01	NumSlave	RW	0x0004 (4)
2001:0	ModbusConfigSlave0	RO	> 10 <
2001:01	Id	RW	0x00E8 (232)
2001:02	NumColis	RW	0x0012 (18)
2001:03	ColisAdress	RW	0x000A (10)
2001:04	NumSwitches	RW	0x0003 (3)
2001:05	SwitchesAdress	RW	0x271A (10010)
2001:06	NumInRegisters	RW	0x0006 (6)
2001:07	InRegistersAdress	RW	0x753A (30010)
2001:08	NumOutRegisters	RW	0x0004 (4)
2001:09	OutRegistersAdress	RW	0x9C4A (40010)
2001:0A	PollingTimeMs	RW	0x00002710 (10000)
2002:0	ModbusConfigSlave1	RO	> 10 <
2003:0	ModbusConfigSlave2	RO	> 10 <
2004:0	ModbusConfigSlave3	RO	> 10 <

Figure 3.1: *ODs of configuring slave device*

The index 2000h is the quantity of slave devices connected to the gateway currently.

The index 2001h-2004h is the configuration information of each slave device.

The meaning of each sub-index is as:

ID of that slave device, the quantity of coils, address of first coil, the number of discrete input, address of first discrete input, the quantity of the input registers, the address of first input register, the number of the holding registers, the address of first holding register, and the time of polling of this device.

For example, as shown in the picture above, the significance of each sub-index is that: The number of Modbus slave device connected to the gateway now is 4, the ID of device 0 is 232(E8h), it has 18 coils, 3 discrete inputs, 6 input registers and 4 holding registers. their address in order:10,10010,30010,40010.

3.1.2 Configure serial communication

2020:0	UartConfigUart1	RO	> 4 <
2020:01	BaundRate	RW	0x0001C200 (115200)
2020:02	BitCountPerChar	RW	0x0000 (0)
2020:03	ParityMode	RW	0x0000 (0)
2020:04	StopBitCount	RW	0x0000 (0)

Figure 3.2: *ODs of configuring serial communication*

See *figure*3.2, the index 2020h is used to configure the serial communication The meaning of each sub-index is as:

The baud rate, it's equal to the real value of the baud rate used, the default value is 115200.

The number of data bits per UART character, we use 0 to define the number is 8, 1 as 9. And the default value is 0.

The UART parity mode options, we use 0 to define the mode as parity disabled, 2 as type even, and 3 as type odd. And the default value is as 0.

The number of stop bits to configure. We use 0 to define it as one stop bit 1 as two stop bits.

For example, as shown in the picture above, the significance of each sub-index is that: The baud rate for the serial port is 115200bps, the number of data bits per UART character is 8, the UART parity mode is disabled, and the number of stop bits is 1 bit.

3.2 Update and enable the configuration

2005:0	ConfigUpdateAndCompletedFlag	RO	> 2 <
2005:01	ConfigUpdateFlag	RW	0x0000 (0)
2005:02	ConfigUpdateCompletedFlag	RW	0x0001 (1)

Figure 3.3: *ODs of updating and enabling configuration*

The default value for the two sub-index is 0 and can both set and reset by the EtherCAT master. When the users have finished configuring, we can set the first sub-index 2005h.0, *figure*3.3.

When the slaver, the gateway, has detected the value is set, it will update the configuration and enable and begin the communication between the gateway and the Modbus slave devices. And then the gateway will set the value of the second sub-index 2005h.02 to tell the EtherCAT master and also the user that the configuration has been completed. After that, the gateway will reset the first sub-index 2005h.01.

For example, as shown in the picture above, the significance of each sub-index is that: The configuration is currently completed, so it's no need to configure it again.

In the process of configuring, the gateway also need to calculate where are those data sent to and collected from Modbus devices are saved, I call it the physical address here to distinguish it from the address that we used in Modbus protocol that we have seen before, In the following, I use all physical addresses to refer to the storage location of data in the gateway cache array.

After calculation, the gateway will update the OD 2011h-201xh, *figure*3.4, whose significance is the storage location. From those index, the EtherCAT master and users can know the significance of data sent to and collected from slave devices.

2011:0	PhyAdressRegistersModbusSlave0	RO	> 10 <
2011:01	StartInputPhyAdress	RO	0x0000 (0)
2011:02	InputColisPhyAdress	RO	0x0000 (0)
2011:03	InputSwitchesPhyAdress	RO	0x0003 (3)
2011:04	InputInRegistersPhyAdress	RO	0x0004 (4)
2011:05	InputOutRegistersPhyAdress	RO	0x0010 (16)
2011:06	EndInputPhyAdress	RO	0x0017 (23)
2011:07	StartOutputPhyAdress	RO	0x0000 (0)
2011:08	OutputColisPhyAdress	RO	0x0000 (0)
2011:09	OutputOutRegistersPhyAdress	RO	0x0003 (3)
2011:0A	EndOutputPhyAdress	RO	0x000A (10)
2012:0	PhyAdressRegistersModbusSlave1	RO	> 10 <
2013:0	PhyAdressRegistersModbusSlave2	RO	> 10 <
2014:0	PhyAdressRegistersModbusSlave3	RO	> 10 <

Figure 3.4: *ODs of physical address*

The meaning of each sub-index for the initial input address, coil input address, discrete status input address, the input register input address, the holding registers input address, the end of the input address, start output address, the coil output address, the holding registers output address, the input address is the corresponding position of the data in the input buffer array and the output address is the corresponding position of the data in the output buffer array.

For example, the above figure means that the starting position of slave device 0 in the input buffer is 0, the starting input position of its coil is 0, the starting position of the discrete state variable is 3, the starting input position of the input register is 4, Hold register initial output position is 16, the end position of slave device 0 at input buffer is 23; the beginning position for slave device 0 in output buffer area is 0, coil start output position is 0, holding

register start output location is 3, and the end position for slave device 0 at the output buffer is 10.

3.3 Read the data in TPDO of the gateway to get data from slave devices

The data saved in TPDO is data read from slave devices.

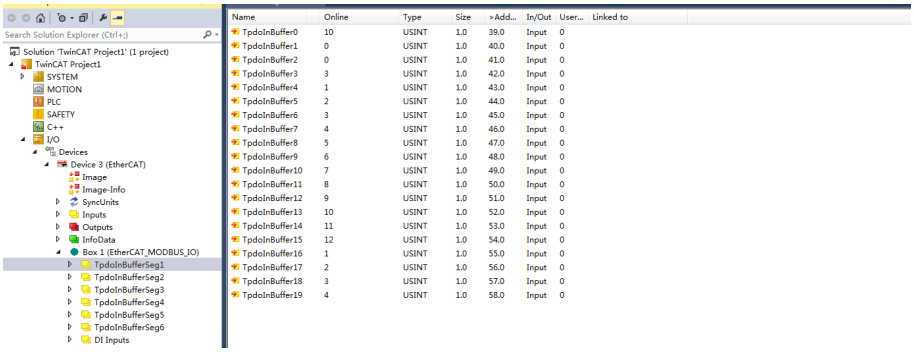

Figure 3.5: *ODs of TPDO, saving data mapped from input buffer*

For example, see *figure*3.5, the significance of these data in this picture is:

Values of coils the slave device 0, 3 bytes, are 10 0 0. They mean that the coil No.2 and No.4 of slave device 0 are switches on and others are switched off.

Values of discrete inputs of slave device 0, 1 byte, is 3, meaning that the status for discrete inputs No.1 and No.2 are switched on and the third one is switched off.

Values of inputting registers of slave device 0 are: 1,2,3,4,5,6,7,8,9,10,11,12, meaning that values for six input registers are followed by 0x0102, 0x0304, 0x0506, 0x0708, 0x090a and 0x0b0c.

Values of holding registers of device 0 (only 4 bytes are shown in the picture) are 1,2,3,4. It means that the values for the first two holding registers are 0x0102, 0x0304.

3.4 Send the instruction of writing to slave devices par the gateway

Users use the EtherCAT master to change the values of RPDO to write new values in slave devices.

For example, see *figure*3.6, in this picture, those data mean that:

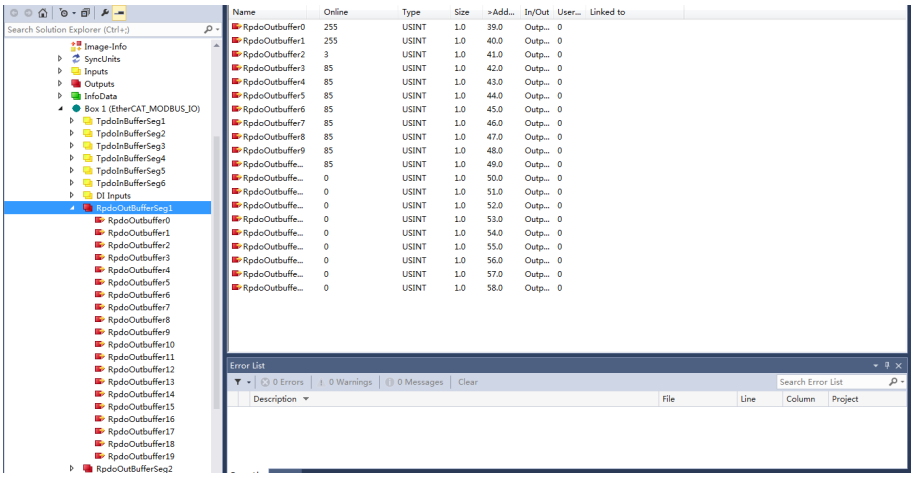

Figure 3.6: *ODs of TPDO, saving data mapped to output buffer*

Set the values of coils of slave device 0, 3 bytes, as 255, 255,3, meaning that switch on all 18 coils of device 0.

Set the values of holding registers of slave device 0, 8 bytes, as 85,85,85,85, 85,85,85,85, meaning that set values of all 4 holding registers as 0x5555.

3.5 Change the configuration when using

If we need to change the configuration when using, we need to reset the flag of configuration completed, OD 2005h.02, to disable the communication between gateway and slave devices.

Chapter 4

Realizations of the function of gateway

4.1 Principle of realization of the function

See *figure*4.1, the EtherCAT Master configure the gateway and slave devices par SDO. What to configure includes the number of slave devices connected, the number of coils, discrete inputs, input registers, and output registers of each slave device, the start address of coils, discrete inputs, input registers, and output registers of each slave device.

The gateway polls each slave device periodically, and the period depends on the tenth sub-index in OD 200xh, the PollingTimeMs.

Before polling, gateway sends the Modbus instruction 08h, diagnostic, to judge whether the device works normally. But in this edition, we haven't added this function in to the code.

After diagnostic, if the gateway hasn't received the right feedback Modbus frame, it will tell the EtherCAT master the status of current slave device is not normal and skip polling this device. If the right feedback Modbus frame gas been received, the gateway will send instruction 01h, 02h, 03h, and 04h to read values of coils, discrete inputs, input registers and holding registers of current device. And then update those values of corresponding OD of TPDO, from which the master can get information of that device.

After polling one device, the gateway will compare the values of coils and holding registers with the values of corresponding OD of TPDO. If there are differences, it means that the values of coils and holding registers need to be updated. Then the gateway will update those dates in output buffer and send instruction 05h, 06h, 10h, 0Fh to write coils and holding registers of that device. And then diagnose and polling the next one.

Users use the EtherCAT master to change the values of RPDO to write new values in slave devices.

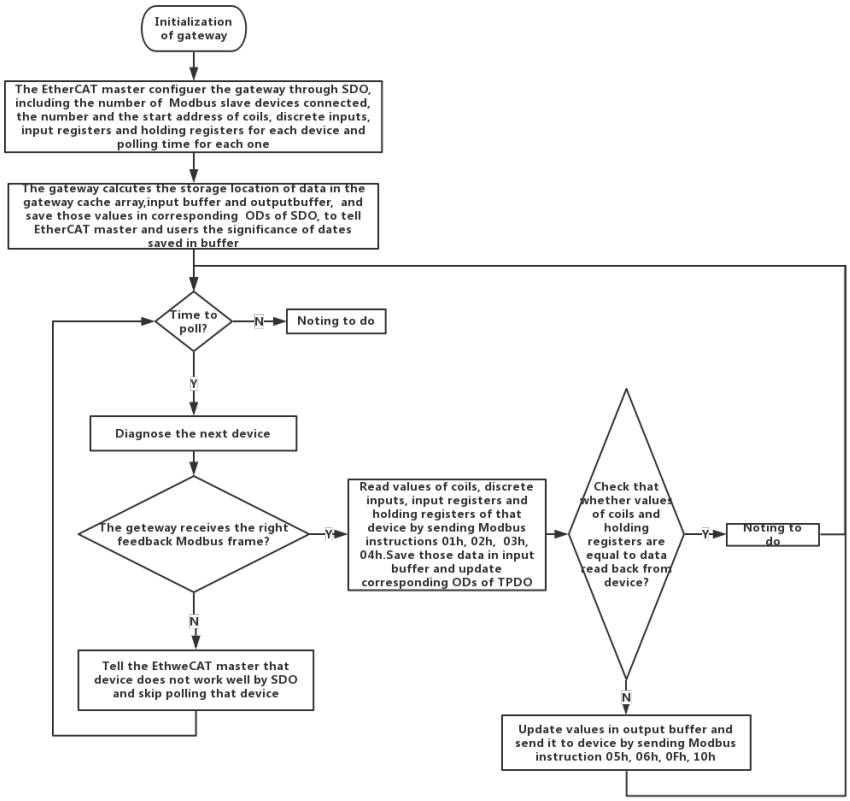

Figure 4.1: *Picture to show the principle of realization of the function*

4.2 Principle of communication

See *figure*4.2.

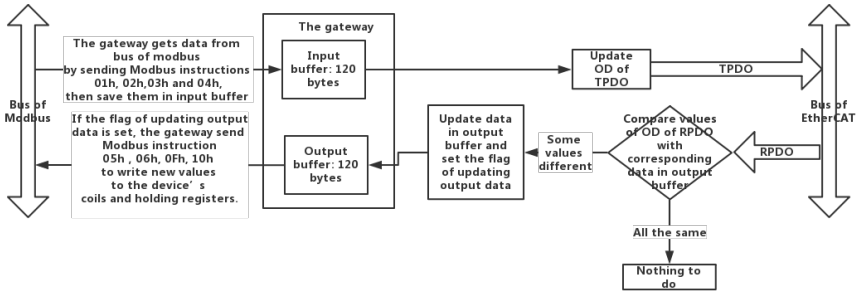

Figure 4.2: *Picture to show the principle of communication*

4.3 Files related to realize the function of gateway

See *table*4.1.

Start	Description
EC_Cfg.h	The driver layer and protocol layer related parameters configuration for the basic functions
Application(.c/.h)	Application layer object dictionary definition and process data processing related functions. The MODBUS protocol related objects dictionaries are defined in MODBUS (.c / .h). Only need to add those ODs to the array of object dictionary, ApplicationObjDic, in Application.c
Events(.c/.h)	Interrupt-related operating functions include GPIO interrupts, LpTimer interrupts, pitTimer1 interrupts, and serial port receive interrupts
main.c	The main function of the project includes system initialization and main function
modbus(.c/.h)	MODBUS protocol processing functions and the definition of related variables
coeappl(.c/.h)	Common object dictionary definition and COE application related operation function
ecat_def.h	Common data types and macro definitions of commonly used standard c functions
ecatappl(.c/.h)	EtherCAT Slave Application-layer related functions, the Interfaces contain PDI, sync0 and sync1 interrupt service functions
ecatcoe(.c/.h)	COE service function interface
ecatslv(.c/.h)	EtherCAT Slave state machine related operation function interface
emcy(.c/.h)	Emergency related operation function interface
mailbox(.c/.h)	Mailbox data processing function interface
objdef(.c/.h)	Object dictionary related operations function interface
sdoserv(.c/.h)	SDO response related operation function interface
9252_HW(.c/.h)	LAN9252 hardware initialization and related register manipulation functions
esc.h	Macro definition of address and mask offset related to LAN9252
SPIDriver(.c/.h)	SPI bus driver function

Table 4.1: *Table of files related to realize the function of gateway*

4.4 Configuration of object dictionary

The index 2000h, See *table*4.2.is one basic object for the gateway, it tells that how many slave devices are connected.

2000h	**Configuration for number of slave devices connected**		
00h	Number of sub-index	16bits	ro
01h	Number of slave devices	16bits	rw
Remarks: The default value for number of slave devices is 4, this value is one basic to configure the others. If this value is less than 4, there is no need to change code of the gateway			

Table 4.2: *Table of description of OD 2000h*

The indexes 2001h-2004h, See *table*4.3.are configurations for all slave devices.

200xh	**Information of configuration for slave devices**		
00h	Number of sub-index	16bits	ro
01h	ID of device	16bits	rw
02h	Number of coils	16bits	rw
03h	Address of the first coil	16bits	rw
04h	Number of discrete inputs	16bits	rw
05h	Address of the first discrete coil	16bits	rw
06h	Number of input registers	16bits	rw
07h	Address of the first input register	16bits	rw
08h	Number of holding registers	16bits	rw
09h	Address of the first holding register	16bits	rw
0Ah	ID of device	32bits	rw
Remarks: It's the same to configure for each Modbus slave device. When the number of device is more than 4, what need to do is just to add object of configuration and add them to the array of object dictionary, ApplicationObjDic, in Application.c. The polling time refers to the interval from the polling of the device to the polling end of the previous slave device, which should be longer than the PDO data update period			

Table 4.3: *Table of description of OD 2001-2004h*

The index 2005h, See *table*4.4.is the flag of needing to update configuration and of configuration having been completed.

Indexes 2011h-2014h, See *table*4.5.is the storage location of data in the gateway cache array, I call them physical addresses. Input address is the position of data in input buffer, output address is position of data in output buffer.

The object 2020h,See *table*4.6.is used to configure serial communication.

Index 2021h, See *table*4.7.is the dictionary entry corresponding to the gateway input

2005h	**Two flags of configuration**		
00h	Number of sub-index	16bits	ro
01h	Flag of needing to update	16bits	rw
02h	Flag of configuration having been completed	16bits	rw
Remarks: The two flags are initialized as 0. When users have completed to configure the OD 2000h-200xh, x is the quantity of devices, by EtherCAT master. The master will set the flag of needing to update. When EtherCAT slave detects the first flag is set, the gateway will configure slave devices, serial ports. After that, the EtherCAT slave will clear data in buffer, set the second flag and reset the first flag and enable communication between gateway and slave devices. If the second flag isn't set, there isn't any communication between gateway and salve devices. When using, if configuration needs to be changed, we need to reset the second flag to stop communication between gateway and devices. When new configuration is done, the master set the first flag to enable new configuration			

Table 4.4: *Table of description of OD 2005h*

201xh	**Physical address of all kinds of data of each slave device**		
00h	Number of sub-index	16bits	ro
01h	Start input address	16bits	rw
02h	Input address of the first data of coils	16bits	rw
03h	Input address of the first data of discrete input	16bits	rw
04h	Input address of the first input register	16bits	rw
05h	Input address of the first holding register	16bits	rw
06h	End of input address	16bits	rw
07h	Start of output address	16bits	rw
08h	Output address of the first data of coils	16bits	rw
09h	Output address of the first data of holding registers	16bits	rw
0Ah	End of output address	16bits	rw
Remarks: Physical addresses are calculated with information given by 2000h-2004h, for EtherCAT master, it's read only. The storage address is consistent with the sub-index of the T / RPDO corresponding to the input / output buffer, and the master can obtain the meaning of the data in the T / RPDO according to those values. What needs to be calculated is the same for each slave device. When number of devices larger than 4. We need to add objects defined in modbus.c, and add those new objects to ApplicationObjDic in Application.c.			

Table 4.5: *Table of description of OD 2011-2014h*

2020h	Configuration for serial communication		
00h	Number of sub-index	16bits	ro
01h	The baud rate	16bits	rw
02h	The number of data bits per UART character	16bits	rw
03h	The UART parity mode options	16bits	rw
04h	The number of stop bits	16bits	rw
Remarks: The baud rate, it's equal to the real value of the baud rate used, the default value is 115200. The number of data bits per UART character, we use 0 to define the number is 8, 1 as 9. And the default value is 0. The UART parity mode options, we use 0 to define the mode as parity disabled, 2 as type even, and 3 as type odd. And the default value is as 0. The number of stop bits to configure. We use 0 to define it as one stop bit 1 as two stop bits.			

Table 4.6: *Table of description of OD 2020h*

buffer and mapped to TPDO (1A00h-1BFFh)

2021h	Objects corresponding to input buffer		
00h	Number of sub-index	16bits	ro
01h - 78h	Array of input buffer	16bits	rw
Remarks: The data stored in the input buffer group is data that the gateway reads from the slave device. The order of the data is consistent with the input address (sub-index 1-6 of the dictionary entry 201xh) calculated from the device configuration information when the gateway is initialized. The master station and users can get meaning of the data according to the data in the array position.			

Table 4.7: *Table of description of OD 2021h*

Index 2022h, See *table*4.8.is the dictionary entry corresponding to the gateway output buffer and mapped to RPDO (1600h-17FFh)

2021h	Objects corresponding to input buffer		
00h	Number of sub-index	16bits	ro
01h - 78h	Array of output buffer	16bits	rw
Remarks: The data stored in the output buffer group is the data that the master station sends to the slave device through the gateway. The order of the data is consistent with the output address (sub-index 7-10 of the dictionary entry 201xh) calculated with the device configuration information when the gateway is initialized, the master changes the data in this array to change the value of the corresponding coils or holding registers.			

Table 4.8: *Table of description of OD 2022h*

4.5 The main macro definitions involved in the implementation of gateway function

File: modbus.h, *table*4.9

Macro definitions	Description
MAX_NUM_ MODBUS_SLAVE	The maximum number of slaves supported by the gateway, which should be greater than the number of slaves currently connected by the gateway represented by sub-key index 1 of the dictionary entry 2000h
MAX_LEN_ OUTPUT_BUFFER	Output buffer maximum data length, the data should be consistent with the dictionary entry 2022h sub-index number
MAX_LEN_ INPUT_BUFFER	Input buffer maximum data length, the data should be consistent with the dictionary entry 2021h sub-index number
MAX_LENGTH_ OF_MODBUS_DATA	Modbus maximum length of effective data, excluding Id, function code and CRC check value, which is estimated by the user to estimate the data frame length of the slave device connected with the gateway to decide whether to modify or not. The current setting is 100, which supports maximum number of registers, input registers or hold registers ,47, or maximum number of coils or discrete inputs,760. That value usually does not need to be modified
MAX_NUM_OF_ REGISTERS_ PER_SLAVE	The maximum number of registers, input registers or holding registers, supported by a single slave, which is based on MAX_LENGTH_OF_MODBUS_DATA, and the sub-index 6, 8 of the dictionary entry 200xh should be less than this value
MAX_NUM_OF_ REGISTERS_ PER_SLAVE	The maximum number of discrete inputs or coils supported by a single slave that is based on MAX_LENGTH_OF_MODBUS_DATA and the sub-index 2, 4 of the dictionary entry 200xh should be less than this value
EnableUart1Pin	Serial port transmit / receive enable pin, which connects to 485 transceivers enable pin
MODBUS_GAP_ TIMEOUT_VAL	MODBU protocol data frame received value, unit: ms, the time interval between two characters of the data frame is greater than this value is considered start of a new frame of data

Table 4.9: *Table of main macro definitions involved*

4.6 The definition of the main data structure type involved in the implementation of function of gateway function

File: modbus.h[6], *table*4.10

Structure type name	Description
MODBUS_COMMON_ FRAME_RTU	MODBUS_RTU common data frame format
MODBUS_0F_ 10_FRAME_RTU	The structure of frame used to writ multiple hold registers or coils
MODBUS_READ_ FRAME_RTU	The structure of frame used to read data from slave devices
MODBUS_READBACK _FRAME_RTU	The structure of frame of feedback of reading instruction
MODBUS_WRITEBACK _FRAME_RTU	The structure of frame of feedback of writing instruction
MODBUS_CHECK_ AND_CHECKBACK_RTU	The structure of frame of diagnostics and diagnostic feedback (not yet used)
CONFIG_INFO_GLOBAL_ MODBUS_PER_SLAVE	The structure used to describe the general information of slave devices, including the slave input /output buffer start/end physical address
The structure corresponds to dictionary entry	
CONFIG_INFO_ MODBUS_PER_SLAVE	The structure used to describe the information of each slave device, corresponding to OD 200xh
PHY_ADRESS_OF_ REGISTERS_PER_SLAVE	The structure used to describe physical address of data of each slave device, corresponding to OD 201xh
UART_CONFIG	The structure used to describe the information of configuring serial communication, corresponding to OD 2020h
MODBUS_SLAVE_ CONFIG	The structure used to describe the number of slave devices, corresponding to OD 2000h
CONFIG_UPDATE_AND_ COMPLETED_FLAG	The structure used to describe flags of updating configuration and configuration completed, corresponding to OD 2005h
OBJECT_INPUT_ BUFFER	The structure used to describe input buffer, corresponding to OD2021h, mapping to TPDO
OBJECT_OUTPUT_ BUFFER	The structure used to describe output buffer, corresponding to OD 2022h, mapping to RPDO

Table 4.10: *Table of main data structure types involved*

4.7 The function of implementation the function of gateway

File: modbus.c, *table*4.11

void **PackModbusReadInsWithOnlyId**(UINT16 SlaveId)
It is called in function **ReadAndWriteModbusSlave**(), which is in **MainLoop**(), to determine whether the configuration is completed when polling time arrives. If the configuration is completed, it sends polling instruction according to the serial number of device
void **CheckAndUpdateOutDataAfterPolling** (UINT16 SlaveId)
It's called in function **ReadAndWriteModbusSlave** (), which is in **MainLoop**(). After instructions of polling is send, it will check whether it's needed to update data in output buffer. If needed, it packs instruction of writing according to data in output data and send them to devices
void **AnalyticUartPacket**(UINT8 CharRev)
The function is used to analyze data received from serial port, called in the serial port receive interrupt. It analyzes whether the data frame is valid. If so, it will set the flag of Modbus feedback completing. When the function of sending Modbus, instruction detects that the flag is set, it exits the block
void **AnalyticModbusRevFrame** (MODBUS_READBACK_FRAME_RTU* pModebusRevDataFrame)
The function of analyzing the frame of Modbus, called in **AnalyticUartPacket**(). If frame received is judged to be valid in **AnalyticUartPacket**(), the frame will be passed to **AnalyticModbusRevFrame**() and effective data will be saved in input buffer, those corresponding dictionary entries will be updated

Table 4.11: *Table of main functions involved*

4.8 The detailed description of some important functions

4.8.1 The function of sending modbus

File: modbus.c

Reading instruction, multiply writing instruction, and singly writing instruction are included.

void **UartSendSyncWrIns** (MODBUS_0F_10_FRAME_RTU *pModebusSyncWrFrame)

Function: Sending Modbus multiply writing instruction

Actual parameter: **pModebusSyncWrFrame**, the pointer of frame of multiply writing

to be send

void **UartSendSingleWrIns** (MODBUS_05_06_FRAME_RTU *pModbusSingleWrFrame)

Function: Sending Modbus singly writing instruction

Actual parameter: **pModbusSingleWrFrame**, the pointer of frame of singly writing to be send

void **UartSendSyncReadIns** (MODBUS_READ_FRAME_RTU *pModbusSyncReadFrame)

Function: Sending Modbus reading instruction, no difference between singly and multiply

Actual parameter: **pModbusSyncReadFrame**, the pointer of frame of reading to be send

Those three functions above are used to calculate the CRC check value according to the data frame to be sent and write it to the temporary array, DataToSend, and send it out through the serial port.

Before sending, I close the global interrupt, and then pass all data into FIFO in once, realizing by the function **UART_SendDataWithoutPolling**(). When finished, reopen the global interrupt.

Closing and reopening the global interrupt is aimed to avoid sending Modbus instruction be interrupted, which will possibly lead to the interval between two characters in one frame bigger than 1.5-character times and be judged to be wrong by slave devices.

After FIFO is empty, it will delay 1.5-character times, realized by the function **Delay1_5CharDecrement**(). This delay is aimed at waiting the last character is send through RS485 and then disable 485 transceivers to enable 485 receivers.

After that, delay for another 2-character times to make sure that no character is send in 3.5-character time.

After sending, reset the flag of Modbus response completing flag, **ModbusFeedbackTimeOutFlag**, reset the flag of timeout of response, **ModbusInsFeedBackCompletedFlag**, and reset the counter for response time, **ModbusFeedbackCounterMs**. Then, polling those two flags, **ModbusInsFeedBackCompletedFlag** and **ModbusFeedbackTimeOutFlag**, until response is completed or timeout.

4.8.2 The function of packing modbus instruction

Function of packing frame of reading instruction, of multiply writing coil instruction, of multiply writing holding register instruction, of singly writing coil are included. Among those functions, the two functions of multiply writing function works as sending frame after

packing, so it's needed to pass the pointer of frame having been packed to that function and then send it to device. For those functions of singly writing, it's needed to send instruction for several times, so I realize the function of sending inner the function of sending. Once frame is packed, it is send immediately.

Function of packing instruction of reading

void **PackModbusReadInsWithIdCate** (UINT16 SlaveId, category_of_modbus_registers_t Cate, MODBUS_READ_FRAME_RTU* pModbusSyncReadFrame)

Function: Packing Modbus instruction of reading according to the serial number of device and the kind of data to read, but not send.

Actual parameter: **SlaveId**, the serial number of device, noting that slave serial number is only the slave number, and its Modbus Id, the two are different.

Cate, the category of data to be read, kCoils represents the coil class, kSwitches represents the discrete input, kInRegisters is the input register, kOutRegisters is the holding register class

pModbusSyncReadFrame: pointer of frame of reading to be send.

Remarks: This function is called in **PackModbusReadInsWithOnlyId** (), and it sends instruction to read data of coils, discrete inputs, input registers and output registers in turn. Inner that function, it gets ID of devices according to configurations for devices, calls **GetFuctionWithCate**() to get function code, calls **GetAdressWithIdCate**() to get corresponding address, and calls **GetNumOfRegWithIdCate**() to get the number of data that to be read. The checksum value is calculated when sent.

Function of packing instruction of multiply writing

Functions of packing instruction of multiply writing coils and holding registers are included.

void **PackModbusSyncWrOutRegisters**(UINT16 SlaveId, MODBUS_0F_10_FRAME_RTU* pModbusSyncWrOutRegistersFrame)

Function: Packing instruction of multiply writing holding registers

Actual parameter: **SlaveId**, the serial number of device pModbusSyncWrOutRegisters-Frame, the pointer of frame of multiply writing holding registers to be sent

void **PackModbusSyncWrCoils**(UINT16 SlaveId, MODBUS_0F_10_FRAME_RTU* pModbusSyncWrCoilsFrame)

Function: Packing instruction of multiply writing coils

Actual parameter: **SlaveId**, the serial number of device

pModbusSyncWrCoilsFrame, the pointer of frame of multiply writing coils to be sent

Remarks: Called in **PackModbusWrInsWithIdCate**() and **PackModbusWrInsWithOnlyId**(), when packing finished, instructions is to be sent.

Inner that function, it gets ID of devices, address and number of data according to configuration of devices, and gets data from output buffer.

Position of data in buffer is calculated by calling function **CalBufferPhyAdress**() and saved in OD PhyAdressRegistersModbusSlave. When needing to read data from or write data to buffer, the value of position is both based on that OD.

Function of packing instruction of singly writing

Functions of packing instruction of singly writing holding register and coil are included.

void **PackModbusSinglecWrOutRegisters**(UINT16 SlaveId) Function: Packing instruction of singly writing holding register

Actual parameter: **SlaveId**, the serial number of device

void **PackModbusSinglecWrCoils**(UINT16 SlaveId) Function: Packing instruction of singly writing coil Actual parameter: **SlaveId**, the serial number of device

Remarks: Called in **CheckAndUpdateOutDataAfterPolling**() after checking whether it's needed to update output data. When data needs to be updated, pack instruction and then send it to devices.

Inner the function, it gets ID and address from configuration of devices, and gets data to output in output buffer. It's easy to singly write holding registers, get data from output buffer and send them to devices directly. For coils, it's more complexed, since the data for 8 coils are saved in one-byte data in buffer. So, we need to call **GetCoilsStatusWithBitsCount**() to check the status of current coil is ON or OFF, then to call **ModbusSinglecWrCoilsWithValPerBit**() to send instruction to write a single coil.

Function of calculating physical address

void **CalBufferPhyAdress**(UINT16 SlaveId) Function: Calculating the position of data saved in buffer, physical address, according to configuration of devices Actual parameter: **SlaveId**, the serial number of device

Remarks: The calculation result of this function is the basis of the physical meaning of the data recognized by the EtherCAT master and users. Currently, the storage mode is tight

storage. All types of data of all devices are closely packed with no gaps. Generally, the buffer area is sufficient. The cache is divided into blocks, with different types of data, for each MODBUS device giving a fixed data space so that it is easier for the EtherCAT master to get the meaning of the data.

Chapter 5

Adding the maximum number of slave devices supported by the gateway

5.1 Change macro definition in file, modbus.h

```
//the maximum number of effective data in one Modbus frame
#define MAX_LENGTH_OF_MODBUS_DATA   100
//the maximum number of holding registers for one device supported by gateway, 47 here
#define MAX_NUM_OF_OUTREGISTERS_PER_SLAVE   ((((UINT16)MAX_LENGTH_OF_MODBUS_DATA)-5)>>1)
//the maximum coils supported for one device supported by gateway,760 here
#define MAX_NUM_OF_COLIS_PER_SLAVE  ((((UINT16)MAX_LENGTH_OF_MODBUS_DATA)-5)<<3)
//the maximum number of slave devices supported, 4 as default value
#define MAX_NUM_MODBUS_SLAVE 4
//the maximum length of input buffer, 120 as default value
#define MAX_LEN_INPUT_BUFFER 120
//the maximum length of output buffer, 120 as default value
#define MAX_LEN_OUTPUT_BUFFER 120
```

Figure 5.1: *Picture to show macro definitions need to be changed*

See *figure*5.1, We should change the macro definition MAX_NUM_MODBUS_SLAVE, if devices connected to the gateway more than 4. If the length of buffer to save data is not enough, we need also to change the macro definition MAX_LEN_INPUT_BUFFER and MAX_LEN_OUTPUT_BUFFER. If length of buffer is changed, we need to change the according OD in the XML file.

5.2 Add according definition of OD in file, modbus.c

5.2.1 Add the number of information of global configuration for device

See *figure* 5.2

```
//the list of global physical address for each device, whose number can be expanded. Default number:4
CONFIG_INFO_GLOBAL_MODBUS_PER_SLAVE PhyAdressGlobalModbusSlave[]={
        {0,0,0,0},
        {0,0,0,0},
        {0,0,0,0},
        {0,0,0,0}
};
```

Figure 5.2: *Picture to show the definition of ODs of global configuration*

5.2.2 Add the according OD of configuration

See *figure* 5.3

```
CONFIG_INFO_MODBUS_PER_SLAVE ConfigInfoModbusSlave[]={
        {10,232,18,10,3,10010,6,30010,4,40010,10000},
        {10,208,10,20,16,10020,2,30020,8,40020,10000},
        {10,184,8,30,5,10030,5,30030,6,40030,10000},
        {10,160,23,40,7,10040,1,30040,2,40040,10000},
};
OBJCONST UCHAR OBJMEM aNameConfigInfoModbusSlave0[] = "Slave0 StartInPhyAd\000InColisPhyAd\000InSwitchesPhyAd\000InInRegPhyAd\000
OBJCONST UCHAR OBJMEM aNameConfigInfoModbusSlave1[] = "Slave1 StartInPhyAd\000InColisPhyAd\000InSwitchesPhyAd\000InInRegPhyAd\000
OBJCONST UCHAR OBJMEM aNameConfigInfoModbusSlave2[] = "Slave2 StartInPhyAd\000InColisPhyAd\000InSwitchesPhyAd\000InInRegPhyAd\000
OBJCONST UCHAR OBJMEM aNameConfigInfoModbusSlave3[] = "Slave3 StartInPhyAd\000InColisPhyAd\000InSwitchesPhyAd\000InInRegPhyAd\000
```

Figure 5.3: *Picture to show definition of ODs of configuration*

5.2.3 Add the according OD of physical address

See *figure* 5.4

```
PHY_ADRESS_OF_REGISTERS_PER_SLAVE PhyAdressRegistersModbusSlave[]={
        {10,0,0,0,0,0,0,0,0,0,0},
        {10,0,0,0,0,0,0,0,0,0,0},
        {10,0,0,0,0,0,0,0,0,0,0},
        {10,0,0,0,0,0,0,0,0,0,0}
};
OBJCONST UCHAR OBJMEM aNamePhyAdressRegModbusSlave0[] = "Slave0 Id\000NumColis\000ColisAdress\000NumSwitches\000SwitchesAdress\00
OBJCONST UCHAR OBJMEM aNamePhyAdressRegModbusSlave1[] = "Slave1 Id\000NumColis\000ColisAdress\000NumSwitches\000SwitchesAdress\00
OBJCONST UCHAR OBJMEM aNamePhyAdressRegModbusSlave2[] = "Slave2 Id\000NumColis\000ColisAdress\000NumSwitches\000SwitchesAdress\00
OBJCONST UCHAR OBJMEM aNamePhyAdressRegModbusSlave3[] = "Slave3 Id\000NumColis\000ColisAdress\000NumSwitches\000SwitchesAdress\00
```

Figure 5.4: *Picture to show definition of ODs of physical address*

5.2.4 Add those new OD into the list of ODs, ApplicationObjDic, in Application.c and changing according XML

See *figure* 5.5

```
{NULL,NULL,  0x2000, {DEFTYPE_RECORD, 1 | (OBJCODE_REC << 8)}, asEntryUartModbusConfig, aNameModbusConfig, &ModbusConfig, NULL, NULL, 0x0000
/* Object 0x2001 ConfigInfoModbusSlave0*/
{NULL,NULL,  0x2001, {DEFTYPE_RECORD, 10 | (OBJCODE_REC << 8)}, asEntryConfigInfoModbusSlave, aNameConfigInfoModbusSlave0, &ConfigInfoModbus
/* Object 0x2002 ConfigInfoModbusSlave1*/
{NULL,NULL,  0x2002, {DEFTYPE_RECORD, 10 | (OBJCODE_REC << 8)}, asEntryConfigInfoModbusSlave, aNameConfigInfoModbusSlave1, &ConfigInfoModbus
/* Object 0x2003 ConfigInfoModbusSlave2*/
{NULL,NULL,  0x2003, {DEFTYPE_RECORD, 10 | (OBJCODE_REC << 8)}, asEntryConfigInfoModbusSlave, aNameConfigInfoModbusSlave2, &ConfigInfoModbus
/* Object 0x2004 ConfigInfoModbusSlave3*/
{NULL,NULL,  0x2004, {DEFTYPE_RECORD, 10 | (OBJCODE_REC << 8)}, asEntryConfigInfoModbusSlave, aNameConfigInfoModbusSlave3, &ConfigInfoModbus
/* Object 0x2011 PhyAdressRegistersModbusSlave0*/
{NULL,NULL,  0x2011, {DEFTYPE_RECORD, 10 | (OBJCODE_REC << 8)}, asEntryPhyAdressOfRegistersPerSlave, aNamePhyAdressRegModbusSlave0, &PhyAdre
/* Object 0x2012 PhyAdressRegistersModbusSlave1*/
{NULL,NULL,  0x2012, {DEFTYPE_RECORD, 10 | (OBJCODE_REC << 8)}, asEntryPhyAdressOfRegistersPerSlave, aNamePhyAdressRegModbusSlave1, &PhyAdre
/* Object 0x2013 PhyAdressRegistersModbusSlave2*/
{NULL,NULL,  0x2013, {DEFTYPE_RECORD, 10 | (OBJCODE_REC << 8)}, asEntryPhyAdressOfRegistersPerSlave, aNamePhyAdressRegModbusSlave2, &PhyAdre
/* Object 0x2014 PhyAdressRegistersModbusSlave3*/
{NULL,NULL,  0x2014, {DEFTYPE_RECORD, 10 | (OBJCODE_REC << 8)}, asEntryPhyAdressOfRegistersPerSlave, aNamePhyAdressRegModbusSlave3, &PhyAdre
/* Object 0x2020 UartConfigUart1*/
{NULL,NULL,  0x2020, {DEFTYPE_RECORD, 4 | (OBJCODE_REC << 8)}, asEntryUartConfigUart1, aNameUartConfigUart1, &UartConfigUart1, NULL, NULL, 0
/* Object 0x2021 ObjectInBuffer TPDOMAPPING*/
{NULL,NULL,  0x2021, {DEFTYPE_UNSIGNED8, 120 | (OBJCODE_ARR << 8)}, asEntryObjectInBuffer, aNameObjectInBuffer, &ObjectInBuffer, NULL, NULL,
/* Object 0x2022 ObjectOutBuffer RPDOMAPPING*/
{NULL,NULL,  0x2022, {DEFTYPE_UNSIGNED8, 120 | (OBJCODE_ARR << 8)}, asEntryObjectOutBuffer, aNameObjectOutBuffer, &ObjectOutBuffer, NULL, NU
/* Object 0x2005 ConfigUpdateAndCompletedFlag*/
{NULL,NULL,  0x2005, {DEFTYPE_RECORD, 2 | (OBJCODE_REC << 8)}, asEntryConfigUpdateAndCompletedFlag, aNameConfigUpdateAndCompletedFlag, &Conf
```

Figure 5.5: *Picture to show list of ODs*

Chapter 6

Test for the gateway

6.1 Test with the virtual Modbus slaves

For the common Modbus device, it has four types of data, coil, discrete input, input register and holding register. For the discrete input and input register, they are read only. For the other two kinds, they can be both written and read. During the internship, I have two kinds of devices to test the gateway. One is the Variable-frequency Drive, VFD, Schneider Ativar 71, and the other one is the digital displacement sensors, Panasonic SC-HG1-485. For those two devices, they have only one kind of data, holding register. Besides, I can't test the gateway with multiple devices connected to the gateway in the same time. To test all functions of the gateway, I have used the virtual Modbus[4] to make it work. The software can be download in http://www.modbustools.com/download.html.

6.1.1 Information of virtual Modbus devices

I have four virtual Modbus slave devices in the software, each one has all types of data. We set the polling time for each one is 1000ms, *table*6.1.

So, when configure the gateway by EtherCAT master, Twin CAT 3.1 developed by Beckhoff, and slave device as *figure*6.1:

Serial number	ID	Number of coils	Address of the first coil	Number of discrete inputs	Address of the first discrete input
0	232	18	10	3	10010
1	208	10	20	16	10020
2	184	8	30	5	10030
3	160	23	40	7	10040

Number of input registers	Address of the first input register	Number of holding registers	Address of the first holding register
6	30010	4	40010
2	30020	8	40020
5	30030	6	40030
1	30040	2	40040

Table 6.1: *Table of information of virtual Modbus devices*

2001:0	ConfigInfoModbusSlave0		> 10 <
2001:01	u16Id	RW	0x00E8 (232)
2001:02	u16NumColis	RW	0x0012 (18)
2001:03	u16ColisAdress	RW	0x000A (10)
2001:04	u16NumSwitches	RW	0x0003 (3)
2001:05	u16SwitchesAdress	RW	0x271A (10010)
2001:06	u16NumInRegisters	RW	0x0006 (6)
2001:07	u16InRegistersAdress	RW	0x753A (30010)
2001:08	u16NumOutRegisters	RW	0x0004 (4)
2001:09	u16OutRegistersAdress	RW	0x9C4A (40010)
2001:0A	u32PollingTimeMs	RW	0x000003E8 (1000)
2002:0	ConfigInfoModbusSlave1		> 10 <
2002:01	u16Id	RW	0x00D0 (208)
2002:02	u16NumColis	RW	0x000A (10)
2002:03	u16ColisAdress	RW	0x0014 (20)
2002:04	u16NumSwitches	RW	0x0010 (16)
2002:05	u16SwitchesAdress	RW	0x2724 (10020)
2002:06	u16NumInRegisters	RW	0x0002 (2)
2002:07	u16InRegistersAdress	RW	0x7544 (30020)
2002:08	u16NumOutRegisters	RW	0x0008 (8)
2002:09	u16OutRegistersAdress	RW	0x9C54 (40020)
2002:0A	u32PollingTimeMs	RW	0x000003E8 (1000)
2003:0	ConfigInfoModbusSlave2		> 10 <
2003:01	u16Id	RW	0x00B8 (184)
2003:02	u16NumColis	RW	0x0008 (8)
2003:03	u16ColisAdress	RW	0x001E (30)
2003:04	u16NumSwitches	RW	0x0005 (5)
2003:05	u16SwitchesAdress	RW	0x272E (10030)
2003:06	u16NumInRegisters	RW	0x0005 (5)
2003:07	u16InRegistersAdress	RW	0x754E (30030)
2003:08	u16NumOutRegisters	RW	0x0006 (6)
2003:09	u16OutRegistersAdress	RW	0x9C5E (40030)
2003:0A	u32PollingTimeMs	RW	0x000003E8 (1000)
2004:0	ConfigInfoModbusSlave3		> 10 <
2004:01	u16Id	RW	0x00A0 (160)
2004:02	u16NumColis	RW	0x0017 (23)
2004:03	u16ColisAdress	RW	0x0028 (40)
2004:04	u16NumSwitches	RW	0x0007 (7)
2004:05	u16SwitchesAdress	RW	0x2738 (10040)
2004:06	u16NumInRegisters	RW	0x0001 (1)
2004:07	u16InRegistersAdress	RW	0x7558 (30040)
2004:08	u16NumOutRegisters	RW	0x0002 (2)
2004:09	u16OutRegistersAdress	RW	0x9C68 (40040)
2004:0A	u32PollingTimeMs	RW	0x000003E8 (1000)

Figure 6.1: *Configuration for slaves*

6.1.2 Physical address of each device

When configuration of devices is finished, the physical address is calculated automatically. We can compare the two values, one calculated by excel as the standard answer , *table* 6.2, one calculated by the gateway, *figure*6.2. They are the same.

Serial number	**Start input physical address**	**Start input physical address of coils**	**Start input physical address of discrete inputs**	**Start input physical address of input registers**	**Start input physical address of holding registers**
0	0	0	3	4	16
1	24	24	26	28	32
2	48	48	49	50	60
3	72	72	75	76	78

End input physical address	**Start output physical address**	**Start output physical address of coils**	**Start output physical address of holding registers**	**End output physical address**
23	0	0	3	10
47	11	11	13	28
71	29	29	30	41
3	42	42	45	48

Table 6.2: *The theoretical input & output physical address*

2011:0	PhyAdressRegistersModbusSlave		> 10 <
2011:01	u16StartInputPhyAdress	RO	0x0000 (0)
2011:02	u16InputColisPhyAdress	RO	0x0000 (0)
2011:03	u16InputSwitchesPhyAdress	RO	0x0003 (3)
2011:04	u16InputInRegistersPhyAdress	RO	0x0004 (4)
2011:05	u16InputOutRegistersPhyAdress	RO	0x0010 (16)
2011:06	u16EndInputPhyAdress	RO	0x0017 (23)
2011:07	u16StartOutputPhyAdress	RO	0x0000 (0)
2011:08	u16OutputColisPhyAdress	RO	0x0000 (0)
2011:09	u16OutputOutRegistersPhyAdress	RO	0x0003 (3)
2011:0A	u16EndOutputPhyAdress	RO	0x000A (10)
2012:0	PhyAdressRegistersModbusSlave		> 10 <
2012:01	u16StartInputPhyAdress	RO	0x0018 (24)
2012:02	u16InputColisPhyAdress	RO	0x0018 (24)
2012:03	u16InputSwitchesPhyAdress	RO	0x001A (26)
2012:04	u16InputInRegistersPhyAdress	RO	0x001C (28)
2012:05	u16InputOutRegistersPhyAdress	RO	0x0020 (32)
2012:06	u16EndInputPhyAdress	RO	0x002F (47)
2012:07	u16StartOutputPhyAdress	RO	0x000B (11)
2012:08	u16OutputColisPhyAdress	RO	0x000B (11)
2012:09	u16OutputOutRegistersPhyAdress	RO	0x000D (13)
2012:0A	u16EndOutputPhyAdress	RO	0x001C (28)
2013:0	PhyAdressRegistersModbusSlave		> 10 <
2013:01	u16StartInputPhyAdress	RO	0x0030 (48)
2013:02	u16InputColisPhyAdress	RO	0x0030 (48)
2013:03	u16InputSwitchesPhyAdress	RO	0x0031 (49)
2013:04	u16InputInRegistersPhyAdress	RO	0x0032 (50)
2013:05	u16InputOutRegistersPhyAdress	RO	0x003C (60)
2013:06	u16EndInputPhyAdress	RO	0x0047 (71)
2013:07	u16StartOutputPhyAdress	RO	0x001D (29)
2013:08	u16OutputColisPhyAdress	RO	0x001D (29)
2013:09	u16OutputOutRegistersPhyAdress	RO	0x001E (30)
2013:0A	u16EndOutputPhyAdress	RO	0x0029 (41)
2014:0	PhyAdressRegistersModbusSlave		> 10 <
2014:01	u16StartInputPhyAdress	RO	0x0048 (72)
2014:02	u16InputColisPhyAdress	RO	0x0048 (72)
2014:03	u16InputSwitchesPhyAdress	RO	0x004B (75)
2014:04	u16InputInRegistersPhyAdress	RO	0x004C (76)
2014:05	u16InputOutRegistersPhyAdress	RO	0x004E (78)
2014:06	u16EndInputPhyAdress	RO	0x0051 (81)
2014:07	u16StartOutputPhyAdress	RO	0x002A (42)
2014:08	u16OutputColisPhyAdress	RO	0x002A (42)
2014:09	u16OutputOutRegistersPhyAdress	RO	0x002D (45)
2014:0A	u16EndOutputPhyAdress	RO	0x0030 (48)

Figure 6.2: *Physical address for devices*

6.1.3 Test for data input and output

The picture below, *figure*6.3, shows how test works. The laptop works as the Modbus slave devices with software, Modbus Slave, running on it. And the desktop works as EtherCAT master with Twin CAT running on it.

There are four virtual Modbus slave devices run in the laptop, and for each device, the polling time is 1s. For each device, we read all four types of data first and those data was saved in input buffer and then mapped to TPDO of the gateway. We check that whether values of ODs of TPDO are equal to them shown in the laptop, if so, we judge that the gateway

Figure 6.3: *The picture to show the test with virtual Modbus slave devices*

has read back the right values from devices. Then we change the value in output buffer by changing the value of ODs of RPDO, which will be mapped to the output buffer periodically. Then we wait for the values of coils and holding registers being changed according to values of ODs of RPDO. If it costs the time less than one period, we judge that the gateway has well writing data to devices. In the next period, those values of coils and holding registers saved in ODs of TPDO needs to be same as corresponding values in ODs of TPDO, which means that it's been well read again. Those pictures and tables following is the results of four virtual slave devices.

Those data in table are the theoretical values of devices. Each device has two table to show theoretical values. The first one is to present the process of reading, we change data of discrete inputs and input registers, then comparing them with data read back by the gateway. The second one is to show the process of writing, we change data in output buffer by the EtherCAT master, then compare them with corresponding values of coils and output registers. In the next period, we also compare the values in output buffer with corresponding input buffer.

Device 0

Table of theoretical values, see *table*6.3 and *table*6.4. Figures of results of tests, see *figure*6.4, *figure* 6.5, *figure* and *figure* 6.6

Serial number	Category of data	Bytes of data	Value of data
0	Discrete input	1(3 bits)	0x07(dec:7 bin:00000111)
	Significance of data: Three discrete inputs of device 0 are all switched on		
	Phenomenon: The 3rd data in Input buffer equal to 7		
	Input register	12(2 bytes for each register)	dec:10 100 20 200 30 300
	Significance of data: The values of six input registers are 10 100 20 200 30 300		
	Phenomenon: The 4th-15th data in input buffer equal to 0 10, 0 100,0 20,0 200,0 30, 1 44		

Table 6.3: *Table of theoretically reading of device 0*

Serial number	Category of data	Bytes of data	Value of data
0	coil	3(2bytes and2 bits)	The 0th-2nd data in output buffer equal to 0xFF 0xFF 0x03(dec:255 255 3 bins:11111111 11111111 00000011)
	Significance of data: To switch on all 18 coils of device 0		
	Phenomenon: All coils of device0 are switched on and the 0th-2nd data in input buffer equal to 255 255 3		
	holding register	8(2 bytes for each register)	The 3rd-10th data in output buffer equal to 0 10,0 100,0 20,0 200
	Significance of data: To set the data of holding registers of device 0 as 10, 100,20,200		
	Phenomenon: The four holding registers of device 1 are set as 10,100,20,200 and the 16th -23rd data in input buffer equal to 0 10,0 100,0 20,0 200		

Table 6.4: *Table of theoretically writing of device 0*

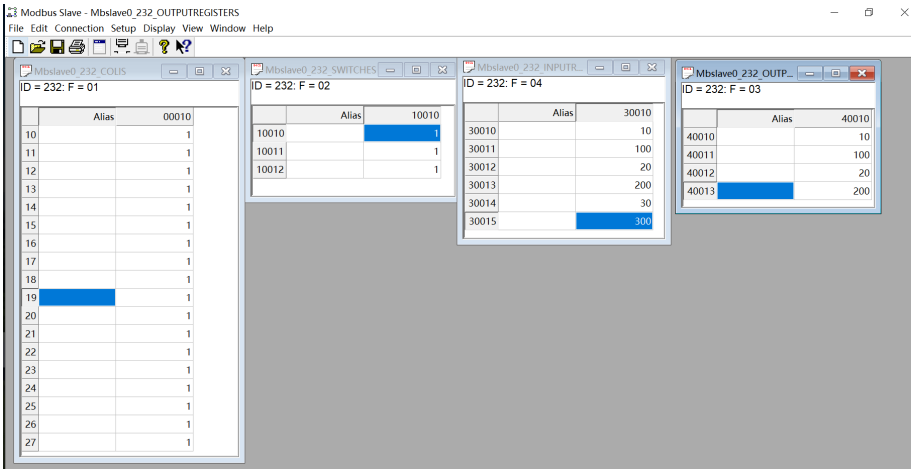

Figure 6.4: *The values of device 0 in virtual Modbus slave software*

Name	Online	Type	Size	>Add...	In/Out	User...	Linked to
InputBuffer0	255	USINT	1.0	39.0	Input	0	
InputBuffer1	255	USINT	1.0	40.0	Input	0	
InputBuffer2	3	USINT	1.0	41.0	Input	0	
InputBuffer3	7	USINT	1.0	42.0	Input	0	
InputBuffer4	0	USINT	1.0	43.0	Input	0	
InputBuffer5	10	USINT	1.0	44.0	Input	0	
InputBuffer6	0	USINT	1.0	45.0	Input	0	
InputBuffer7	100	USINT	1.0	46.0	Input	0	
InputBuffer8	0	USINT	1.0	47.0	Input	0	
InputBuffer9	20	USINT	1.0	48.0	Input	0	
InputBuffer10	0	USINT	1.0	49.0	Input	0	
InputBuffer11	200	USINT	1.0	50.0	Input	0	
InputBuffer12	0	USINT	1.0	51.0	Input	0	
InputBuffer13	30	USINT	1.0	52.0	Input	0	
InputBuffer14	1	USINT	1.0	53.0	Input	0	
InputBuffer15	44	USINT	1.0	54.0	Input	0	
InputBuffer16	0	USINT	1.0	55.0	Input	0	
InputBuffer17	10	USINT	1.0	56.0	Input	0	
InputBuffer18	0	USINT	1.0	57.0	Input	0	
InputBuffer19	100	USINT	1.0	58.0	Input	0	
InputBuffer20	0	USINT	1.0	59.0	Input	0	
InputBuffer21	20	USINT	1.0	60.0	Input	0	
InputBuffer22	0	USINT	1.0	61.0	Input	0	
InputBuffer23	200	USINT	1.0	62.0	Input	0	

Figure 6.5: *Result of data of input buffer of device 0 read by the gateway*

OutputBuffer0	255	USINT	1.0	39.0	Outp...	0
OutputBuffer1	255	USINT	1.0	40.0	Outp...	0
OutputBuffer2	3	USINT	1.0	41.0	Outp...	0
OutputBuffer3	0	USINT	1.0	42.0	Outp...	0
OutputBuffer4	10	USINT	1.0	43.0	Outp...	0
OutputBuffer5	0	USINT	1.0	44.0	Outp...	0
OutputBuffer6	100	USINT	1.0	45.0	Outp...	0
OutputBuffer7	0	USINT	1.0	46.0	Outp...	0
OutputBuffer8	20	USINT	1.0	47.0	Outp...	0
OutputBuffer9	0	USINT	1.0	48.0	Outp...	0
OutputBuffer10	200	USINT	1.0	49.0	Outp...	0

Figure 6.6: *Data of ODs of RPDO, mapped to output buffer of device 0*

Device 1

Table of theoretical values, see *table*6.5 and *table*6.6. Figures of results of tests, see *figure*6.7, *figure* 6.8, *figure* and *figure* 6.9

Serial number	Category of data	Bytes of data	Value of data
1	Discrete input	2(16 bits)	0x55 0x55(dec:85 85 bins:01010101 01010101)
	Significance of data: For those 16 discrete inputs, whose serial number are odd numbers are ON and whose serial number are even numbers are OFF		
	Phenomenon: The 26th-27th data in input buffer are 85,85		
	Input register	4(2 bytes for each register)	dec:100 1000
	Significance of data: The values of two input registers are 100 1000		
	Phenomenon: The 28th-31st data in input buffer equal to 0 100, 3 232		

Table 6.5: *Table of theoretically reading of device 1*

Serial number	Category of data	Bytes of data	Value of data
1	coil	2 (1byte and2 bits)	The 11st-12nd data in output buffer equal to 0xFF 0x55 0x01(dec:85 1 bin:01010101 00000001)
	Significance of data: To switch on all coils of device 1 whose serial number are odd, and to switch off all coils of device 1 whose serial number are even		
	Phenomenon: All coils of device1 whose serial number are odd are switched on and the others are switched off. And the 24th-25th data in input buffer equal to 85 1		
	holding register	16 (2 bytes for each register)	The 13rd-28th data in output buffer equal to 0 100,3 232,0 200,7 208, 1 44, 11 184, 1 144, 15 160
	Significance of data: To set the data of holding registers of device 1 as 100,1000,200,2000,300,3000,400,4000		
	Phenomenon: The eight holding registers of device 1 are set as 100,1000,200,2000,300,3000,400,4000 and the 32nd -47th data in input buffer equal to 0 100,3 232,0 200,7 208, 1 44, 11 184, 1 144, 15 160		

Table 6.6: *Table of theoretically writing of device 1*

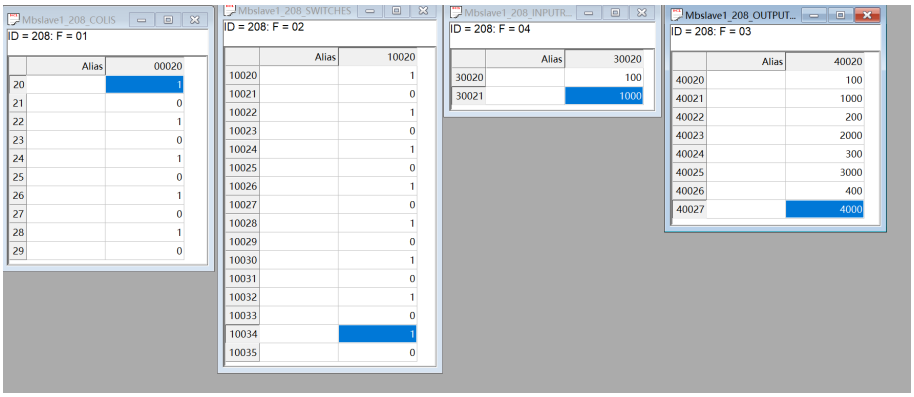

Figure 6.7: *The values of device 1 in virtual Modbus slave software*

Name	Online	Type	Size	>Add...	In/Out	User...	Linked to
InputBuffer24	85	USINT	1.0	63.0	Input	0	
InputBuffer25	1	USINT	1.0	64.0	Input	0	
InputBuffer26	85	USINT	1.0	65.0	Input	0	
InputBuffer27	85	USINT	1.0	66.0	Input	0	
InputBuffer28	0	USINT	1.0	67.0	Input	0	
InputBuffer29	100	USINT	1.0	68.0	Input	0	
InputBuffer30	3	USINT	1.0	69.0	Input	0	
InputBuffer31	232	USINT	1.0	70.0	Input	0	
InputBuffer32	0	USINT	1.0	71.0	Input	0	
InputBuffer33	100	USINT	1.0	72.0	Input	0	
InputBuffer34	3	USINT	1.0	73.0	Input	0	
InputBuffer35	232	USINT	1.0	74.0	Input	0	
InputBuffer36	0	USINT	1.0	75.0	Input	0	
InputBuffer37	200	USINT	1.0	76.0	Input	0	
InputBuffer38	7	USINT	1.0	77.0	Input	0	
InputBuffer39	208	USINT	1.0	78.0	Input	0	
InputBuffer40	1	USINT	1.0	79.0	Input	0	
InputBuffer41	44	USINT	1.0	80.0	Input	0	
InputBuffer42	11	USINT	1.0	81.0	Input	0	
InputBuffer43	184	USINT	1.0	82.0	Input	0	
InputBuffer44	1	USINT	1.0	83.0	Input	0	
InputBuffer45	144	USINT	1.0	84.0	Input	0	
InputBuffer46	15	USINT	1.0	85.0	Input	0	
InputBuffer47	160	USINT	1.0	86.0	Input	0	

Figure 6.8: *Result of data of input buffer of device 1 read by the gateway*

OutputBuffer11	85	USINT	1.0	50.0	Outp...	0
OutputBuffer12	1	USINT	1.0	51.0	Outp...	0
OutputBuffer13	0	USINT	1.0	52.0	Outp...	0
OutputBuffer14	100	USINT	1.0	53.0	Outp...	0
OutputBuffer15	3	USINT	1.0	54.0	Outp...	0
OutputBuffer16	232	USINT	1.0	55.0	Outp...	0
OutputBuffer17	0	USINT	1.0	56.0	Outp...	0
OutputBuffer18	200	USINT	1.0	57.0	Outp...	0
OutputBuffer19	7	USINT	1.0	58.0	Outp...	0
OutputBuffer20	208	USINT	1.0	59.0	Outp...	0
OutputBuffer21	1	USINT	1.0	60.0	Outp...	0
OutputBuffer22	44	USINT	1.0	61.0	Outp...	0
OutputBuffer23	11	USINT	1.0	62.0	Outp...	0
OutputBuffer24	184	USINT	1.0	63.0	Outp...	0
OutputBuffer25	1	USINT	1.0	64.0	Outp...	0
OutputBuffer26	144	USINT	1.0	65.0	Outp...	0
OutputBuffer27	15	USINT	1.0	66.0	Outp...	0
OutputBuffer28	160	USINT	1.0	67.0	Outp...	0

Figure 6.9: *Data of ODs of RPDO, mapped to output buffer of device 1*

Device 2

Table of theoretical values, see *table*6.7 and *table*6.8. Figures of results of tests, see *figure*6.10, *figure* 6.11, and *figure*6.12

Serial number	Category of data	Bytes of data	Value of data
2	Discrete input	1(5 bits)	0x15(dec:21 bin:00010101)
	Significance of data: For those 5 discrete inputs, whose serial number are odd numbers are ON and whose serial number are even numbers are OFF		
	Phenomenon: The 49th data in Input buffer equal to 21		
	Input register	10(2 bytes for each register)	dec:0 10 100 1000 10000
	Significance of data: The values of five input registers are 0 10 100 1000 10000		
	Phenomenon: The 50th-59th data in input buffer equal to 0 0,0 10,0 100,3 232,39 16		

Table 6.7: *Table of theoretically reading of device 2*

Serial number	Category of data	Bytes of data	Value of data
2	coil	1(8 bits)	The 29th data in output buffer equal to 0xAA (dec:170 bin:10101010)
	Significance of data: To switch on all coils of device 2 whose serial number are even, and to switch off all coils of device 2 whose serial number are odd		
	Phenomenon: All coils of device2 whose serial number are even are switched on and the others are switched off. And the 48th data in input buffer equal to 170		
	holding register	12(2 bytes for each register)	The 30th-41st data in output buffer equal to 0 0,0 20,0200,7 208,78 32,117 48
	Significance of data: To set the data of holding registers of device 2 as 0,20,200,2000,20000,30000		
	Phenomenon: The six holding registers of device 2 are set as 0,20,200,2000,20000,30000 and the 60th -71st data in input buffer equal to 0 0,0 20,0200,7 208,78 32,117 48		

Table 6.8: *Table of theoretically writing of device 2*

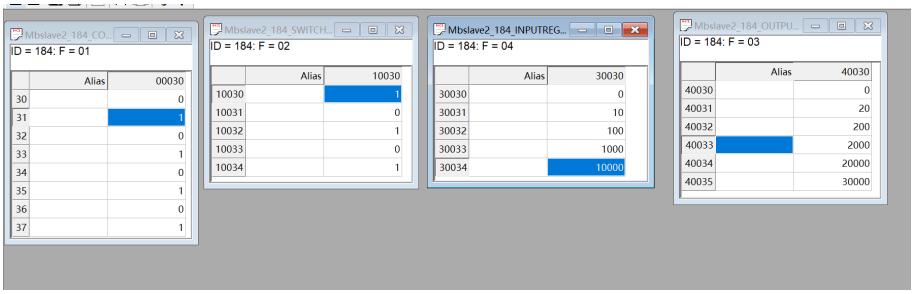

Figure 6.10: *The values of device 2 in virtual Modbus slave software*

Name	Online	Type	Size	>Add...	In/Out	User...	Linked to
InputBuffer48	170	USINT	1.0	87.0	Input	0	
InputBuffer49	21	USINT	1.0	88.0	Input	0	
InputBuffer50	0	USINT	1.0	89.0	Input	0	
InputBuffer51	0	USINT	1.0	90.0	Input	0	
InputBuffer52	0	USINT	1.0	91.0	Input	0	
InputBuffer53	10	USINT	1.0	92.0	Input	0	
InputBuffer54	0	USINT	1.0	93.0	Input	0	
InputBuffer55	100	USINT	1.0	94.0	Input	0	
InputBuffer56	3	USINT	1.0	95.0	Input	0	
InputBuffer57	232	USINT	1.0	96.0	Input	0	
InputBuffer58	39	USINT	1.0	97.0	Input	0	
InputBuffer59	16	USINT	1.0	98.0	Input	0	
InputBuffer60	0	USINT	1.0	99.0	Input	0	
InputBuffer61	0	USINT	1.0	100.0	Input	0	
InputBuffer62	0	USINT	1.0	101.0	Input	0	
InputBuffer63	20	USINT	1.0	102.0	Input	0	
InputBuffer64	0	USINT	1.0	103.0	Input	0	
InputBuffer65	200	USINT	1.0	104.0	Input	0	
InputBuffer66	7	USINT	1.0	105.0	Input	0	
InputBuffer67	208	USINT	1.0	106.0	Input	0	
InputBuffer68	78	USINT	1.0	107.0	Input	0	
InputBuffer69	32	USINT	1.0	108.0	Input	0	
InputBuffer70	117	USINT	1.0	109.0	Input	0	
InputBuffer71	48	USINT	1.0	110.0	Input	0	

Figure 6.11: *Result of data of input buffer of device 2 read by the gateway*

Name	Online	Type	Size	>Add...	In/Out	User...	Linked to
OutputBuffer27	15	USINT	1.0	66.0	Outp...	0	
OutputBuffer28	160	USINT	1.0	67.0	Outp...	0	
OutputBuffer29	170	USINT	1.0	68.0	Outp...	0	
OutputBuffer30	0	USINT	1.0	69.0	Outp...	0	
OutputBuffer31	0	USINT	1.0	70.0	Outp...	0	
OutputBuffer32	0	USINT	1.0	71.0	Outp...	0	
OutputBuffer33	20	USINT	1.0	72.0	Outp...	0	
OutputBuffer34	0	USINT	1.0	73.0	Outp...	0	
OutputBuffer35	200	USINT	1.0	74.0	Outp...	0	
OutputBuffer36	7	USINT	1.0	75.0	Outp...	0	
OutputBuffer37	208	USINT	1.0	76.0	Outp...	0	
OutputBuffer38	78	USINT	1.0	77.0	Outp...	0	
OutputBuffer39	32	USINT	1.0	78.0	Outp...	0	
OutputBuffer40	117	USINT	1.0	79.0	Outp...	0	
OutputBuffer41	48	USINT	1.0	80.0	Outp...	0	

Figure 6.12: *Data of ODs of RPDO, mapped to output buffer of device 2*

Device 3

Table of theoretical values, see *table*6.9 and *table*6.10. Figures of results of tests, see *figure*6.13, *figure* 6.14, *figure* and *figure* 6.15

Serial number	Category of data	Bytes of data	Value of data
3	Discrete input	1(7 bits)	0x7F (dec:127 bin:01111111)
	Significance of data: Seven discrete inputs of device 3 are all switched on		
	Phenomenon: The 75th data in input buffer equal to 127		
	Input register	2(2 bytes for each register)	dec:32767
	Significance of data:		
	Phenomenon: The 76th-77th data in input buffer equal to 127 255		

Table 6.9: *Table of theoretically reading of device 3*

Serial number	Category of data	Bytes of data	Value of data
3	coil	3(2bytes and7 bits)	The 42nd-44th data in output buffer equal to 0xFF 0xFF 0x7F (dec:255 255 127 bin:11111111 11111111 01111111)
	Significance of data: To switch on all 23 coils of device 3		
	Phenomenon: All coils of device 3 are switched on and the 72nd-74th data in input buffer equal to 255 255 127		
	holding register	4(2 bytes for each register)	The 45th-48th data in output buffer equal to 1 44, 117 48
	Significance of data: To set the data of holding registers of device 3 as 300,30000		
	Phenomenon: The two holding registers of device 3 are set as 300,30000 and the 78th -81st data in input buffer equal to1 44, 117 48		

Table 6.10: *Table of theoretically writing of device 3*

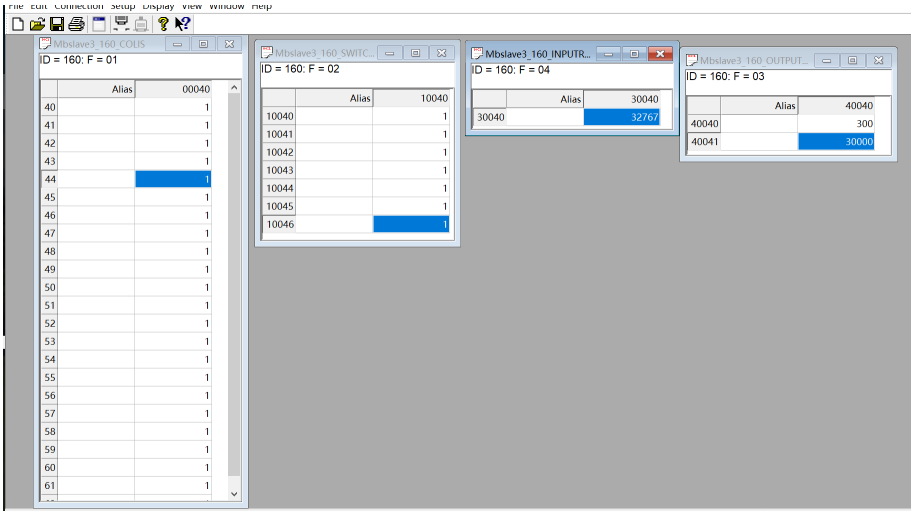

Figure 6.13: *The values of device 3 in virtual Modbus slave software*

Name	Online	Type	Size	>Add...	In/Out	User...	Linked to
InputBuffer72	255	USINT	1.0	111.0	Input	0	
InputBuffer73	255	USINT	1.0	112.0	Input	0	
InputBuffer74	127	USINT	1.0	113.0	Input	0	
InputBuffer75	127	USINT	1.0	114.0	Input	0	
InputBuffer76	127	USINT	1.0	115.0	Input	0	
InputBuffer77	255	USINT	1.0	116.0	Input	0	
InputBuffer78	1	USINT	1.0	117.0	Input	0	
InputBuffer79	44	USINT	1.0	118.0	Input	0	
InputBuffer80	117	USINT	1.0	119.0	Input	0	
InputBuffer81	48	USINT	1.0	120.0	Input	0	

Figure 6.14: *Result of data of input buffer of device 3 read by the gateway*

Name	Online	Type	Size	>Add...	In/Out	User...	Linked to
OutputBuffer42	255	USINT	1.0	81.0	Outp...	0	
OutputBuffer43	255	USINT	1.0	82.0	Outp...	0	
OutputBuffer44	127	USINT	1.0	83.0	Outp...	0	
OutputBuffer45	1	USINT	1.0	84.0	Outp...	0	
OutputBuffer46	44	USINT	1.0	85.0	Outp...	0	
OutputBuffer47	117	USINT	1.0	86.0	Outp...	0	
OutputBuffer48	48	USINT	1.0	87.0	Outp...	0	

Figure 6.15: *Data of ODs of RPDO, mapped to output buffer of device 3*

6.1.4 Conclusion for test with the virtual Modbus slaves

Through those tables and pictures, we can know that the gateway operates well when connected to the EtherCAT master, supported by Twin CAT 3.1, and these four Modbus slave devices, supported by software of virtual Modbus slave devices. I have changed the synchronization cycle of EtherCAT from 1ms to 4ms, have changed the polling time of each devices

from 1ms to 1000ms, and have changed the baud rate of serial communication from 4800bps and 115200bps, the gateway can all completed writing and reading in one polling period. We can conclude that the transformation between the two protocols works well in all conditions that is possible in industry. In the next part, I will test how stable the gateway works with connecting the gateway to the Variable-frequency Drive, VFD, Schneider Ativar 71.

6.2 Test with the real Modbus slaves

6.2.1 Test for the stability of the gateway

To test the stability of the gateway, I connected the gateway to the EtherCAT master, supported by the software Multiprog, and the real Modbus slave, the Variable-frequency Drive, VFD, Schneider Ativar 71. *figure* 6.16 and *figure* 6.17 show how it works.

The VFD connects to the gateway

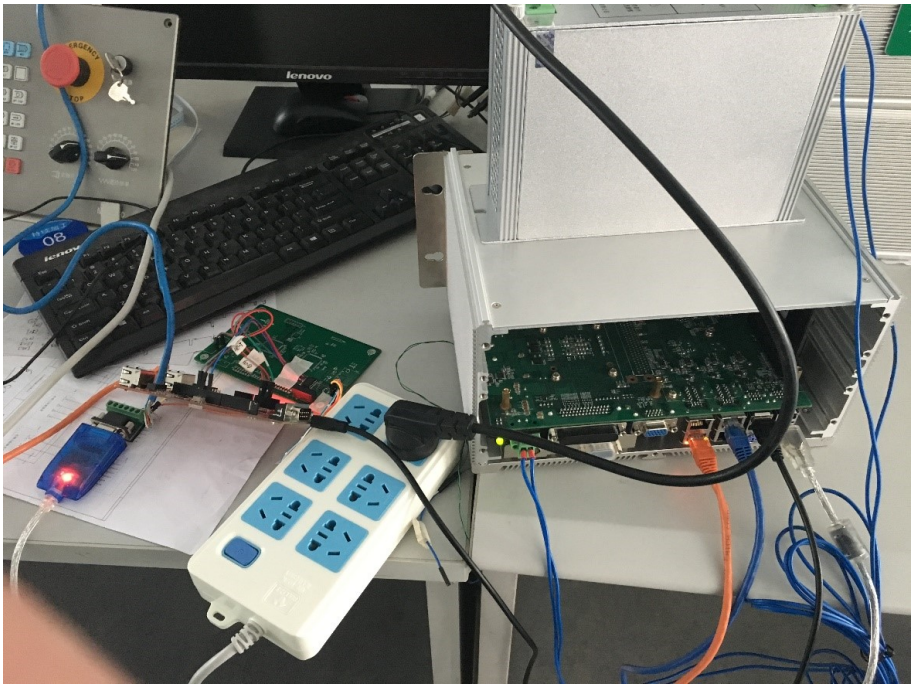

Figure 6.16: *The gateway connects to EtherCAT master, supported by an IPC*

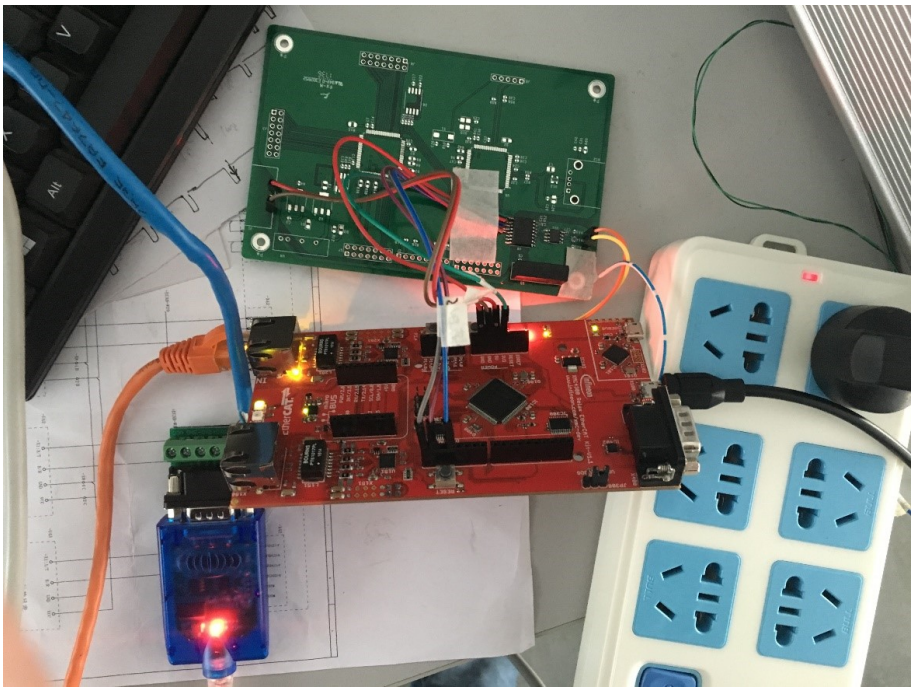

Figure 6.17: *The detailed picture of the gateway*

As having introduced before, the VFD has only one type of data, the holding registers. So, when configuring the gateway, I defined the number of coils, discrete inputs and input registers all zero and choose its four holding registers to test the gateway. The information of that four holding registers are shown in *table* 6.11.

Name	**Address**	**Range**
Smooth coefficient in the start of acceleration, tA1	9005	0-100
Smooth coefficient in the end of acceleration, tA2	9006	0-(100- tA1)
Smooth coefficient in the start of deceleration, tA3	9007	0-100
Smooth coefficient in the end of deceleration, tA4	9008	0-(100- tA3)

Table 6.11: *Presentation of the four holding registers of the VFD*

In the beginning of test, I have tasted the gateway with condition of baud rate from 4800bps to 38400bps, synchronization cycle of EtherCAT from 0.1ms to 1ms, and polling period from 1ms to 2000ms to make sure that it works well in all those conditions possible. For test of stability, I chose the baud rate as 19200bps, synchronization cycle as 1ms, and polling period as 1000ms as the condition, for they are the most used in factory.

In that condition, I used the EtherCAT master to change the values of those four registers

from 0 t0 49. And then detect whether the data read back from those registers are equal to the value expected and begin to time until those values being send are equal to those values being read back. If the master detects that they are equal, it will reset timer and change the value for registers again. If the time costs are more than the maximum time possible, the master will judge it as an error and counter of error adds one.

The most terrible condition is that when the gateway has just finished writing data to device, the master changes the data and the timer begin to time. It's shown in the *figure* 6.18. Those value can only be written in to device in the next period and after another period been read back. The longest time possible is 2.3s. For the timer used in the master is set to add one every one second. So, I set the condition as judging an error taking palace as timer more than three seconds.

Figure 6.18: *The worst condition for execution*

*figure*6.19 is a Presentation of part of codes of EtherCAT master, supported by Multiprog.

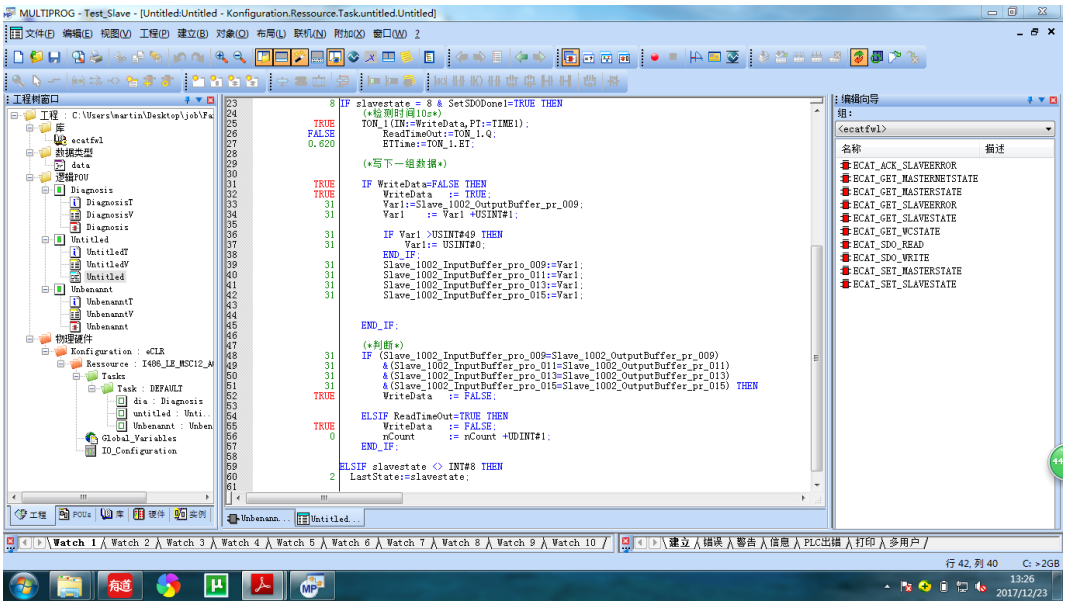

Figure 6.19: *Presentation of part of codes of EtherCAT master, supported by Multiprog*

The test for stability has last for 30 days and no error appears, we can conclude that the stability can satisfy our meets.

Conclusion for test with the real Modbus slaves

Test with the VFD has continued for 30 days, the stability has been verified. But it exits some pities for having just test with one VFD and only with one type of data. Before the gateway to be used in the new system of CNC, it needs to be tested as what I have done with the virtual Modbus devices. It needs to connects to one EtherCAT master and multiple Modbus slave devices with multiple types of data.

Chapter 7

Reflection on the internship

This six-month period is a very special and fresh experience in my life. In this chapter I reflect on the internship. Regarding my learning goals I shortly discuss my experiences. If I have achieved my goal, whether I experienced difficulties and what I think I have to improve ?

The internship is my first time to handle a hard real-time project. The period between each synchronisation is less than 4 ms, and in some case of test, we have tried also a more strict constrain with period of 1 ms. For that reason, the windows of execution should be strictly divided for each process, and besides, details of each function should be well designed. This project provide me therefore a great opportunity to synthesize my knowledge of data structure, of serial communication, of Ethernet industrial, etc.

Another gain important is my improved programming habits. As the gateway that I designed is for the future system, some other engineers will continue to develop it base on my code. It is therefore necessary to make sure the code to be more manageable, more portable, and more robust. To achieve that goal, the habit of version control, modular programming and keeping development journal are insisted on during programming. In contrast, those basic rules are often ignored when in school.

Bibliography

[1] Introduction of beijing jingdiao group co.
http://www.jingdiao.com/en/groupintroduction_en/groupprofile.html.

[2] Introduction of ethercat.
https://www.ethercat.org/en/technology.html.

[3] Introduction of modbus.
https://en.wikipedia.org/wiki/Modbus.

[4] Introduction of modbusvirtual.
http://www.modbustools.com/modbus.html.

[5] Z. L. Holger Zeltwanger. *Fieldbus CANopen design and application*. Beijing University of Aeronautics and Astronautics Press, 2011.

[6] X. Ji. *Design and Application of Industrial Ethernet Fieldbus EtherCAT Driver*. Beijing University of Aeronautics and Astronautics Press, 2010.

[7] L. Zhengjun. *Fieldbus and Industrial Ethernet and its Application Technology*. Machinery Industry Press, 2011.