Design and Implementation of Modbus Slave Based on ARM Platform and FreeRTOS Environment

Tan-Sy Nguyen
Faculty of Electrical and Electronics Engineering
University of Technology
Ho Chi Minh City, Vietnam
Email: ntsy@hcmut.edu.vn

Thai-Hoang Huynh
Faculty of Electrical and Electronics Engineering
University of Technology
Ho Chi Minh City, Vietnam
Email: hthoang@hcmut.edu.vn

***Abstract*— The paper introduces the design of Modbus slave based on Modbus RTU over RS-485 and its implementation. By designing a new software architecture and embedded open stacks, development lifetime is shortened as well as the software is easy to port, maintain, and reuse. A new approach inspired from Modbus functionality makes it easier for developer to divide program into three tasks and synchronize them. An ARM based Modbus slave hardware is built and experiments have been carried out to validate that this Modbus slave is a reliable and stable communication system. It acquires many desired properties of an industrial embedded network software.**

Keywords — Modbus slave; RS-485; software architecture; FreeRTOS; multi-task programming

I. INTRODUCTION

Modbus is an industrial protocol standard that has been in use for many years. Modbus ASCII and Modbus RTU are relatively simple serial protocols that use EIA-232 (RS-232) or EIA-485 (RS-485) to transmit data packets while Modbus TCP exchanges data over Ethernet TCP/IP. The protocol defines function codes and the encoding scheme for transferring data either as single points (1-bit, coils) or as 16-bit data registers. This basic data packet is then encapsulated according to the protocol specifications for Modbus ASCII, RTU, or TCP. Modbus is not only used in industrial like PLC network, but also applied in many embedded systems scaled from simple ([7] uses AT89C52, [6], [10], [11], [12] are based on ARM processor core) to complicated applications ([4] runs on embedded Linux, [5] runs on QNX Neutrino RTOS). FreeMODBUS is an open Modbus protocol stack which is recommended in official website of Modbus organization. It is designed specially for embedded systems. This protocol stack can support most of popular function codes which are used for supervisory control application.

In the past, most embedded systems were scheduled statically by means of a so-called “while true and interrupt event”, often built in a highly system-dependent way. The complexity of modern embedded systems is constantly increasing, as they must perform more sophisticated functions than in the past. These problems make developers always try to find another approach for their software design. Modularity and portability are achieved by using real-time operating system as a base, while the adaption of open-source components maximizes re-usability and minimizes software development cost and time as presented in [8] and [14]. This paper focuses on designing new software architecture and the program execution is coordinated by a real-time operating system. A real-time application is a set of concurrent, cooperating tasks that synchronize and exchange information with each other. ARM based hardware is also designed and implemented to verify the operation and performance of the software.

The paper is structured as following: section II and section III discuss more details about FreeRTOS and FreeMODBUS, then hardware design of the system and some modules are shown in section IV. Section V explains the design of software architecture. In additions, in this section we present a new approach to design program for Modbus slave and implement it based on our software architecture. The result is shown in section VI when this Modbus slave is tested with simulation software and PLC S7-200. Finally, section VII concludes the paper.

II. FREERTOS

A. Overview

FreeRTOS, a mini Real Time Kernel was founded by Richard Barry. Its design has been developed to fit on very small embedded systems and implemented only a very minimal set of functions. Some prominent properties of FreeRTOS are listed as below.

Scalability: It was ported up to 35 hardware architectures, ranging from 8-bit small micro-controllers to full featured 32-bit processors including ARM, MSP430, AVR, PIC, 8051, etc. Depending on these architectures, the kernel can be compiled on 14 different compilers.

Portability: The porting is eased by several factors. Firstly, the FreeRTOS code base is small. It composes a total of three core files and an additional port file needed for the kernel itself.

Secondly, it is mostly written in standard C. Only a few lines of assembly code are necessary to adapt it into a given platform.

Open source: It is licensed under a modified GPL and can be used in commercial applications under this license. FreeRTOS code is freely available on its website, which makes the kernel easy to study and understand.

Simple and dynamic scheduler: FreeRTOS features a Round Robin, priority-based scheduler. Each task is assigned a priority. Highest priority ready-tasks will be executed and tasks with the same priority share the CPU time in a Round Robin fashion.

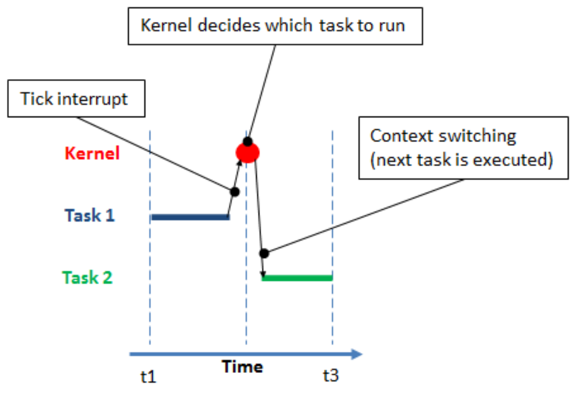

Figure 1 Operation of FreeRTOS scheduler [16]

FreeRTOS allows an unlimited number of tasks to be run as long as the system hardware and memory can handle it. As a real time operating system, FreeRTOS is able to handle both cyclic and acyclic tasks.

B. *Tasks communication and synchronization*

Applications that use RTOS are considered as a set of mini independent programs because each task has its own memory (heap and stack). Consequently, FreeRTOS provides several methods for inter-task communication and synchronization consisting of queue and semaphore. In this paper, we use queue for data exchange between tasks and synchronization.

Queue mechanism can be used in the communications between two tasks or between tasks and Interrupt Service Routine (ISR). A queue is a structure which is able to store and restore data and used as First In First Out (FIFO) buffer.

When a single task reads in a queue, it is moved to "Blocked" state and moved back to "Ready" as soon as data has been written in the queue by another task or an interrupt. If several tasks are trying to read the same queue, the highest priority task will read it first. Finally, if several tasks with the same priority are trying to read, the first task who asked for a read operation is chosen. A task can also specify a maximum waiting time for the queue to allow it to be read. After this time, the task switches back automatically to "Ready" state.

Writing on a queue obeys to the same rules as reading it. When a task tries to write to a queue, it has to wait for it to have some free space: the task is blocked until another task finishes reading the queue and frees some space. If several tasks attempt to write to the same queue, the higher priority task is chosen first. If several tasks with the same priority are trying to write on a queue, then the first one to wait is chosen.

III. MODBUS PROTOCOL AND FREEMODBUS STACK

Modbus is a message based protocol. The Master/Slave method is used and only the node assigned as the Master may initiate a command. On Ethernet, any device can send out a Modbus command, although usually only one master device does so. Modbus message contains device address it intends for. This device will act on the command; other devices might receive it and then skip (an exception is broadcast message sent to node 0 which is acted on without acknowledge).

Modbus messages also contain checksum information, to allow the recipient to detect transmission errors. The serial protocol has the parity check, besides, the ASCII mode uses the LRC check, and the RTU mode uses CRC16 check.

The most important field in Modbus message is function code. It represents action need to perform by slave device and the data field following after this will provide more information about this request. In this paper, we use FreeMODBUS library, a free implementation of the popular Modbus protocol specially targeted for embedded systems. In its newest version, FreeMODBUS provides an implementation of the Modbus Application Protocol v1.1a [1] and supports RTU/ASCII transmission modes defined in the Modbus over serial line specification 1.0 [2], it also supports Modbus TCP defined in Modbus Messaging on TCP/IP Implementation Guide v1.0a. The following functions are currently supported in FreeMODBUS (as description on official website [3]):

- Read Input Register (0x04)
- Read Holding Registers (0x03)
- Write Single/Multiple Register(s) (0x06/0x10)
- Read/Write Multiple Registers (0x17)
- Read Coils (0x01)
- Write Single/Multiple Coil(s) (0x05/0x0F)
- Read Discrete Inputs (0x02)
- Report Slave ID (0x11)

FreeMODBUS stack is divided into several layers as shown in table 1. As a universal model, the realization of the Modbus application layer and the protocol stack is independent of the Microcontroller (MCU) and hardware layout. Only physical layer is hardware-dependent and needs to be ported for each specific platform: ARM, PIC, AVR …

TABLE 1 FREEMODBUS SOURCE (MAPPING WITH OSI MODEL)

Application layer eMBInit, eMBEnable, eMBPoll …
Data link layer eMBRTUStart, eMBRTUStop, eMBRTUSend …
Physical layer Port.c, portserial.c, porttimer.c, portevent.c

IV. HARDWARE DESIGN

This section outlines the hardware of the system being considered in this case study. The whole system mainly includes the following parts: main control, communication, address configuration, isolated input, Darlington output, analog input and analog output.

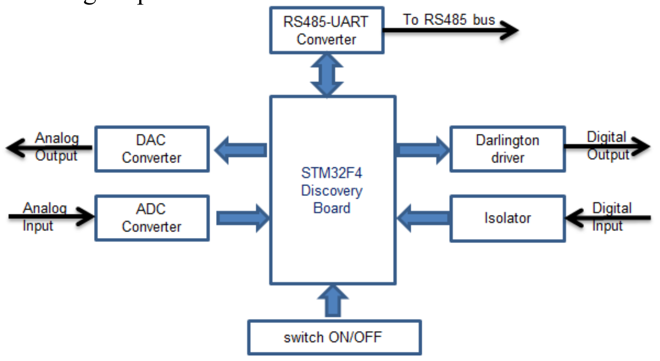

Figure 2 Hardware design of Modbus slave

A. *Main control module*

The main control module is built around STM32F407VG (main microcontroller of STM32 Discovery evaluation kit). It has an ARM Cortex M4 processor core which can run up to 168 MHz. In addition, it extends full primary peripherals for normal applications such as UART, SPI, ADC … and some advanced connectivity peripherals like USB, CAN and Ethernet. It is evaluated as a strong, stable microcontroller and used widely in many systems. STM32F4 Discovery kit also includes debugger for developing and uploading program. Because of its convenience, this board is used to build a demo system in this paper.

B. *Communication module*

There is no microcontroller which can support RS-485 directly. But via a converter circuit using UART – RS-485 transceiver IC, any MCU which supports UART can communicate normally with others on RS 485 network. In this paper, the IC SN75176 of Texas Instrument is used and the schematic is shown in figure 3.

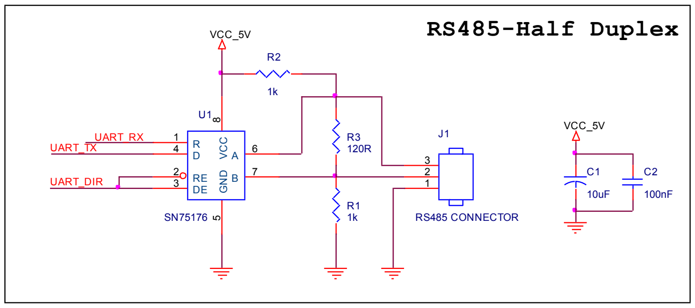

Figure 3 UART RS-485 converter circuit

This circuit is designed to run in half-duplex mode. It lets the system run in receiving mode when UART_DIR pin is pulled down. Otherwise, when UART_DIR pin is pulled up, the system runs in transmitting mode. It is fully compatible with Modbus slave functionality. Normally, a slave device waits for request from the master and only responds if the request messages contain its address.

C. *Address configuration module*

In order to make a new device join network more easily and more flexible, an address configuration module is added to the system. The aim is to setup 4 lower address bits of the device while 4 higher address bits is fixed by 0b1000 (0x8). As the result, up to 16 devices can connect in the same network.

This module simply switches a pin of MCU on/off and the corresponding address bit is obtained as 1 or 0 when the system starts up. Schematic and address table are shown on figure 4 and table 2.

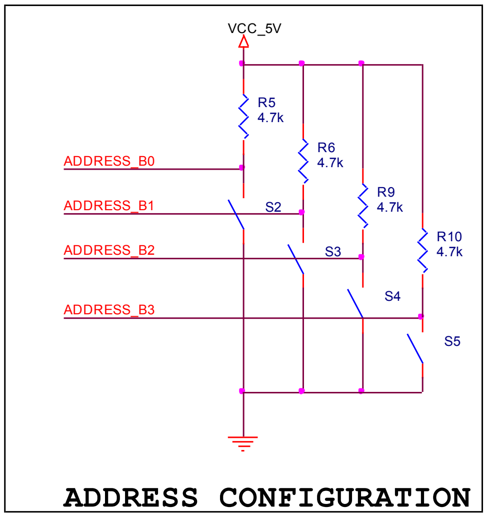

Figure 4 Address configuration module.

TABLE 2 ADDRESS LIST OF MODBUS SLAVE

Bit7…Bit4	Bit3…Bit0	Device address
1000	0000	0b10000000 (0x80)
	…	…
	1111	0b10001111 (0x8F)

V. SOFTWARE DESIGN AND IMPLEMENTATION

A. *Software architecture*

In this section, we focus on designing software architecture and realization. The target of the software is not only a strong interaction to hardware but also flexibility and portability. It makes the system easy to be extended and ported to more powerful hardware to meet future application requirements. Moreover, this software enables users to change MCU or hardware layout easily. The software design uses layers architecture as shown in figure 5.

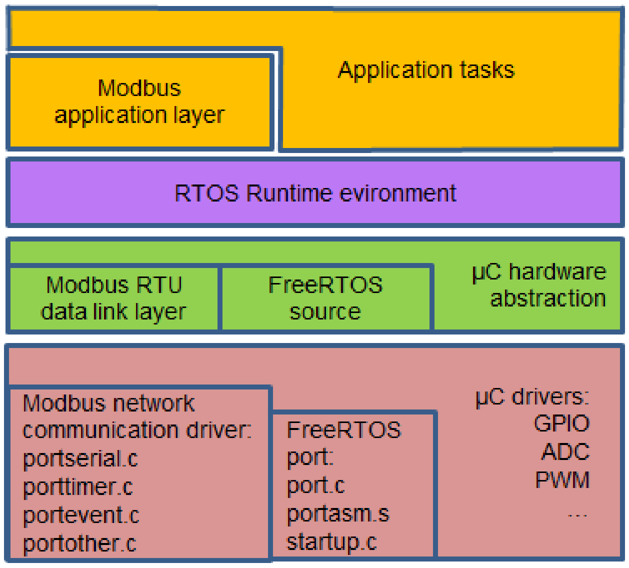
Modbus application layer
Application tasks
RTOS Runtime evironment
Modbus RTU data link layer
FreeRTOS source
µC hardware abstraction
Modbus network communication driver: portserial.c porttimer.c portevent.c portother.c
FreeRTOS port: port.c portasm.s startup.c
µC drivers: GPIO ADC PWM …

Figure 5 Software architecture of system

1) *Microcontroller driver*

Microcontroller driver is the lowest layer of the software. It contains internal drivers or specific drivers which are software modules with directly access to peripherals or memory. Ordinarily, these modules are provided by manufactures. This layer makes upper layers become independent on microcontroller.

2) *Microcontroller hardware abstraction*

This layer interfaces the driver of layer 1 (Microcontroller driver) and offers standard APIs for access to peripherals and devices regardless of their connection to microcontroller (port pins, type of interfaces …). This layer makes higher layers isolated from hardware layout.

3) *Runtime environment*

Runtime environment is a layer which provides communication services to application layer, manages program, and separates application layer from lower layers. In this research, FreeRTOS is chosen because it has enough basic functions such as multi-task execution, tasking synchronization and data exchanging. Moreover, it takes less time to develop program running in FreeRTOS because of its simplicity.

4) *Application layer*

Based on runtime environment, application layer becomes a separated region. Many parts or tasks in this layer communicate with each other by using services from runtime environment. Therefore, developers who implement the programs do not need to have much knowledge about the system hardware as well as other parts of the program. When a big system is developed, programmers can divide it into small parts and develop them independently.

B. *Dividing into multi-task and functionality*

In our approach, Modbus slave program can be split into two sides: network communication side and I/O hardware side, with three small functions or three tasks. Memory is a middle place where both sides communicate to each other. This structure makes the program easy to understand and implement, but it still has one disadvantage that the request from the master needs to wait one period to be updated to hardware. The task dividing or program flow is shown on figure 6.

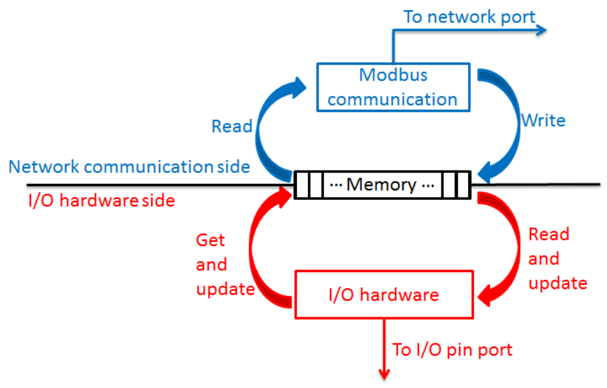

Figure 6 Tasks dividing and relationship

1) *Modbus communication task*

This task handles Modbus interface, receives and analyzes incoming packets, gets information from memory and responds with corresponding data. It does not have a fixed period to execute. This period depends on the duration of sending the message from the master.

2) *I/O hardware task*

This task is used for updating information from/to memory to/from I/O pin port. This operation is implemented in a period of T=10 ms, using timer to ensure that each message sent from the master is replied to and executed immediately.

3) Memory handling task

Modbus slave's memory has bits or registers which are defined in the same way as other industrial devices and shown as table 3. For each type of bits or registers in the memory, the corresponding C-code definition is presented: Coils and Discrete Inputs are simply considered as 1-bit variables, Input Registers and Holding Registers are considered as 16-bit variables (unsigned short data type in C).

TABLE 3 MEMORY MAPPING AND DESCRIPTION OF MODBUS SLAVE

Common name	**Data type/Usage**	**Starting address**
Coils output	Bit, binary value, flag	00001
#define NCOILS 8 *uint8_t ucCoilBuf[NCOILS/8];*		
Discrete Inputs	Bit, binary input	10001
#define NDISCRETE_INPUTS 8 *uint8_t ucDiscInBuf[NDISCRETE_INPUTS /8] ;*		
Input Registers	Analog value	30001
#define NINPUT_REGS 8 *uint16_t usRegInBuf[NINPUT_REGS];*		
Holding Registers	Analog values, variables, parameters …	40001
#define NHOLDING_REGS 8 *uint16_t usRegHoldBuf[NHOLDING_REGS];*		

In single-core system, only one task can be executed at the same time, so memory access collision error never happens. While in multi-core system, this error can occur if two tasks access memory at the same time. Therefore, in single core system there are two ways to exchange data between tasks: using "extern" variables as common memory space and task synchronization techniques like queue and semaphore. Even STM32F407 is a single core microcontroller, queue is still used to synchronize and exchange data between two tasks because it makes us manage memory better and the program can be extended easier in the future. Task synchronization is presented below.

C. Synchronization

Multi-task programming makes program clearer to understand and easier to develop. However, one of the challenges of multi-task programming is task synchronization. This is a basic problem which every developer must not only understand clearly but also solve carefully and smartly. If programmers analyze and sketch out a good plan, the performance of program will be higher and more stable. Otherwise, the program runs slower and sometimes memory fault will happen. The synchronization algorithm can be presented by using sequence diagrams as shown in figure 7 and figure 8:

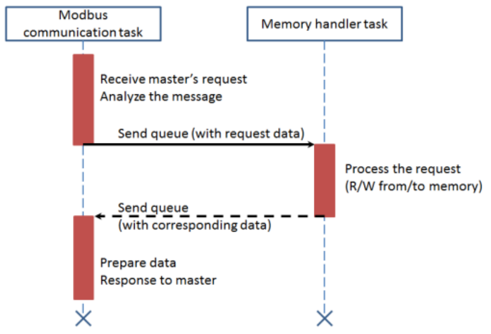

Figure 7 Synchronization between Modbus communication task and Memory handler task

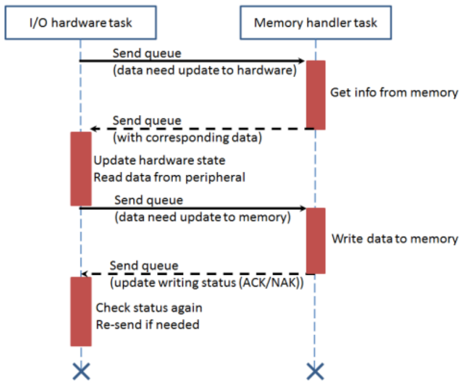

Figure 8 Synchronization between I/O hardware task and Memory handler task

Memory handler task only accepts a request of a task and processes it. Whenever it receives a request from task, it will enter critical section and exit after processing done. If other tasks have request in this duration, they have to queue and wait.

VI. EXPERIMENT AND RESULT

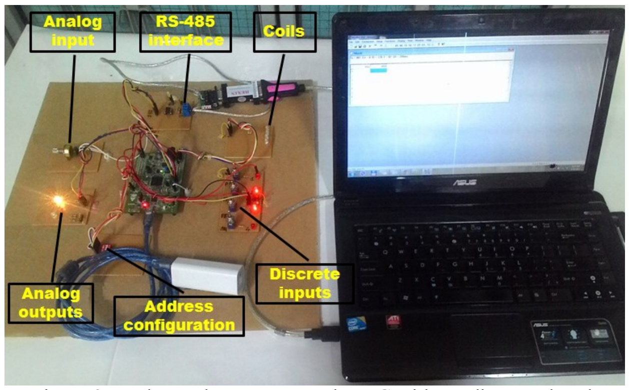

Figure 9 Hardware demo connected to PC with Modbus Pool tool.

1) *Testing with software simulation*

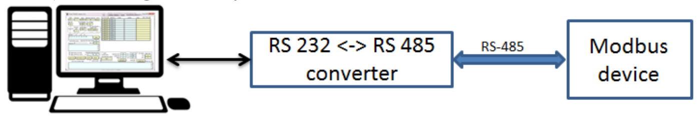

Figure 10 Software verification with Modbus Pool simulation tool

Testing system is set up as shown in Figure 10 in which PC is connected to Modbus slave via a converter circuit. In this paper, Modbus Pool tool is used to simulate Modbus master on PC. This tool sends request every 200 ms to Modbus slave to refresh values of coils, discrete inputs and registers. It is executed continuously every ten minutes for each testing time and the result is shown in table 4 as below.

TABLE 4 TESTING RESULT WITH SIMULATION TOOL.

Request	Response	Successful message	Failure message
Write multi coils: 80 0F 00 01 00 04 01 0F 8A FE	80 0F 00 01 00 04 1B D9	2936	0
Read discrete inputs: 80 02 00 01 00 04 36 18	80 02 01 05 49 B7	2936	0
Read input registers: 80 04 00 01 00 01 7E 1B	80 04 02 09 2C 82 A3	2936	1
Write holding registers: 80 10 00 01 00 03 06 00 00 00 00 00 00 4A 05	80 10 00 01 00 03 CF D9	2936	1

2) *Testing communication with PLC S7-200*

PLC S7-200 supports Modbus RTU serial over RS232. Using a converter circuit as in the test case above, this device can communicate with PLC S7-200 easily. When the connection is established, PLC S7-200 operates as a master and requests this slave to do coils blink. Another example is also implemented as following: PLC S7-200 requests this device to get a discrete input value, and then it sets corresponding value to its output. In these examples, the number of failure message cannot be checked as the test case above but operation of the system is real-time and smooth. User can collect data and control every I/O pin port of the slave normally like other industrial devices.

VII. CONCLUSION

The ARM based industrial system design using FreeRTOS has been deployed successfully in Modbus slave application. The proposed software architecture is portable for any hardware change and can be extended easily for multi-processor system in the future. This architecture is not only used in this application but also applied into any small embedded system which can support RTOS. In any test with simulation tool as well as real PLC hardware, Modbus slave's operation is reliable and stable. This system is a platform for applications in many fields such as automobile, data acquisition, control and so on. It depends on which extended components are integrated. Example: PID or fuzzy controller, FIR/IIR digital filter … They will be studied and embedded into the system in next researches.

REFERENCES

[1] Modbus Organization (2012). Modbus Application Protocol Specification v1.1a [Online]. Available: http://www.modbus.org/docs/Modbus_Application_Protocol_V1_1a.pdf [Jun. 24, 2015].

[2] Modbus Organization (2012). MODBUS over Serial Line Specification & Implementation guide V1.0 [Online]. Available: http://www.modbus.org/docs/Modbus_over_serial_line_V1.pdf [Jun. 24, 2015].

[3] FreeMODBUS stack, http://www.freemodbus.org/

[4] Dao-gang Peng, Hao Zhang, Li Yang and Hui Li, "Design and Realization of Modbus Protocol Based on Embedded Linux System", ICESS Symposia '08. International Conference on Embedded Software and Systems Symposia, pp.275 - 280, July 2008

[5] Sen Xu, Haipeng Pan, Jia Ren and Jie Su, "Design of the Modbus Communication through Serial Port in QNX Operation System", ISECS International Colloquium on Computing, Communication, Control, and Management (CCCM 2008), pp.434 - 438, Aug 2008

[6] Daogang Peng, Hao Zhang, Kai Zhang, Hui Li and Fei Xia, "Research and Development of the Remote I/O Data Acquisition System Based on Embedded ARM Platform", IEEE International Conference on Electronic Computer Technology, pp.341 - 344, Feb 2009.

[7] Bo Cui and Guangbin Xu, "Design and realization of an intelligent data acquisition and display system based on AT89C52 and modbus", ISECS International Colloquium on Computing, Communication, Control, and Management (CCCM 2009), pp.455 - 458, Aug. 2009

[8] Gianluca Cena, Ranieri Cesarato and Ivan Cibrario Bertolotti, "An RTOS-Based Design for Inexpensive Distributed Embedded System", IEEE International Symposium on Industrial Electronics (ISIE), pp.1716 - 1721, July 2010.

[9] Tu Xuyue, "Design of Air Compressor Monitoring System Based on Modbus Protocol", IEEE International Conference on Electrical and Control Engineering (ICECE), pp.710 - 713, June 2010

[10] Bo Qu and Daowei Fan, "Design of remote data monitoring and recording system based on ARM", 2nd IEEE International Conference on Industrial and Information Systems (IIS), pp.252 - 255, July 2010

[11] B.B.Shabarinath and Nidhi Gaur, "MODBUS communication in microcontroller based elevator controller", IEEE International Conference on Control, Automation, Robotics and Embedded Systems (CARE),pp.1 - 5, Dec 2013

[12] Kelong Wang, Daogang Peng, Lei Song and Hao Zhang, "Implementation of Modbus Communication Protocol based on ARM Coretx-M0", IEEE International Conference on System Science and Engineering (ICSSE), pp.69 - 73, July 2014

[13] Gustavo Künzel, Matheus Antônio Corrêa Ribeiro and Carlos Eduardo Pereira, "A tool for response time and schedulability analysis in modbus serial communications", IEEE International Conference on Industrial Informatics (INDIN), pp.446 - 451, July 2014

[14] Ivan Cibrario Bertolotti and Tingting Hu, "Modular design of an open-source, networked embedded system", ScienceDirect, June 2014

[15] Modbus Tools, http://www.modbustools.com/index.html.

[16] Richard Barry, Using the FreeRTOS Real Time kernel: A Practical Guide, http://shop.freertos.org/RTOS_primer_books_and_manual_s/1819.htm