JKSCI

한국컴퓨터정보학회논문지
Journal of The Korea Society of Computer and Information
Vol. 25 No. 6, pp. 1-8, June 2020
https://doi.org/10.9708/jksci.2020.25.06.001

Independent I/O Relay Class Design Using Modbus Protocol for Embedded Systems

Ki-Su Kim*, Jong-Chan Lee*

*Student, School of Computer Information Engineering, Kunsan University, Kunsan, Korea
*Professor, Dept. of Computer Information Engineering, Kunsan University, Kunsan, Korea

[Abstract]

Communication between system modules is applied using the Modbus protocol in industrial sites including smart factories, industrial drones, building energy management systems, PLCs, ships, trains, and airplanes. The existing Modbus was used for serial communication, but the recent Modbus protocol is used for TCP/IP communication.The Modbus protocol supports RTU, TCP and ASCII, and implements and uses protocols in embedded systems. However, the transmission I/O devices for RTU, TCP, and ASCII-based protocols may differ. For example, RTU and ASCII communications transmit on a serial-based communication protocol, but in some cases, Ethernet TCP/IP transmission is required. In particular, since the C language (object-oriented) is used in embedded systems, the complexity of source code related to I/O registers increases. In this study, we designed software that can logically separate I/O functions from embedded devices, and designed the execution logic of each instance requiring I/O processing through a delegate class instance with Modbus RTU, TCP, and ASCII protocol generation. We designed and experimented with software that can separate communication I/O processing and logical execution logic for each instance.

▸ **Key words**: Build Design Pattern, Software Design, Modbus Protocols, Protocol Builder, Avr128, UART

[요 약]

스마트팩토리, 산업용 드론, 빌딩 에너지 관리 시스템, PLC, 선박, 기차 및 비행기를 포함한 산업 현장에서 Modbus프로토콜을 사용하여 시스템 모듈 간 통신을 적용한다. 기존의 Modbus는 시리얼 직렬통신으로 사용되었지만, 최근 Modbus프로토콜은 TCP/IP 통신으로 사용된다. Modbus프로토콜은 3가지 유형의 RTU, TCP 및 ASCII를 지원하고 임베디스 시스템에 프로토콜을 구현 하여 사용 한다. 하지만 RTU, TCP, ASCII기반의 프로토콜은 각 송신 I/O장치가 다를 수 있다. 예를 들어 RTU, ASCII 통신은 시리얼기반으로 통신 프로토콜을 송신하지만 이더넷 TCP/IP 송신을 요구하는 경우도 있다. 이와 같은 문제는 특히 임베디드 시스템에서 C언어(절차 지향)를 사용하기 때문에 I/O레지스터 관련 소스코드의 복잡성 증가 문제가 발생된다. 본 연구는 임베디드 장치에서 I/O함수를 논리적으로 분리 가능한 소프트웨어 설계를 진행하고, 더불어 대리자 클래스 인스턴스를 통하여 I/O처리가 필요한 각 인스턴스의 수행 로직을 Modbus RTU, TCP, ASCII프로토콜 생성으로 설계 하였고 인스턴스별 통신 I/O처리와 논리적 수행 로직을 분리 가능한 소프트웨어 설계와 실험을 하였다.

▸ **주제어**: 빌드 디자인패턴, 소프트웨어 설계, 모드버스 프로토콜, 프로토콜 빌더, 범용 비동기 송수신

• First Author: Ki-Su Kim, Corresponding Author: Jong-Chan Lee
*Ki-Su Kim (vennomnight1@kunsan.ac.kr), School of Computer Information Engineering, Kunsan University
*Jong-Chan Lee (chan2000@kunsan.ac.kr), Dept. of Computer Information Engineering, Kunsan University
• Received: 2020. 03. 26, Revised: 2020. 06. 01, Accepted: 2020. 06. 02.

http://www.ksci.re.kr pISSN:1598-849X | eISSN:2383-9945

I. Introduction

제4차 산업 혁명(Fourth Industrial Revolution, 4IR)은 정보 통신기술(ICT)의 융합으로 이루어낸 혁명 시대를 말한다. 18 세기 초기 산업 혁명 이후 네 번째로 중요한 산업 시대이다. 이 혁명의 핵심은 인공지능, 로봇공학, 사물인터넷, 무인 운송 수단(무인 항공기, 무인 자동차), 3차원 인쇄, 나노 기술과 같은 6대 분야에서 새로운 기술 혁신이다[1]. 사물인터넷(Internet of Things,이하 IOT)은 각종 사물에 센서와 통신기능을 내장 하여 인터넷에 연결하는 기술로서 유, 무선 통신을 통하여 각종 사물을 연결하는 기술을 의미한다[2]. 최근에는 스마트팩토리의 센서 네트워크를 통해 연속적으로 데이터를 수집하고 수집된 빅데이터를 활용하고 있다. 예를 들어 부품의 교체 주기 또는 생산라인의 환경 혹은 생산에 필요한 가스투입량 등 생산제품의 환경요소 설정에 활용되고 있고 기존에는 환경설정을 전문가를 통하여 하였지만 최근에는 수집된 빅데이터를 활용하여 비전문가도 환경요소 설정이 가능하게 되었다. 스마트팩토리에서 기존의 장비와 인터페이스 하여 정보를 수집하는 기술이 필수가 되었고, 현재까지 많은 자동화장비에 PLC(Programmable Logic Controller), HMI(Human Machine Interface)가 주로 사용 되고 있다. 제조업의 스마트팩토리 전환은 전 세계적인 추세이고, 국내 제조업체들도 스마트팩토리 구축에 적극적이다. 기존의 PLC 및 HMI와 같은 기기에서 데이터를 수집하기 위해서는 통신 기술이 핵심이다. 최근에서는 많은 PLC 및 HMI장비에서 이더넷통신 장비를 활용하여 데이터 수집을 하고 있고 이더넷 통신장비가 없는 설비에서는 RS232 또는 RS485시리얼 통신으로 PLC장비 및 HMI로부터 데이터를 수집하고 있다. 모든 통신이 가능한 PLC및 HMI장비는 산업표준프로토콜인 Modbus 프로토콜을 지원하고 있다[3].

따라서 수요가 증가되는 데이터수집 및 처리에서 이더넷 혹은 시리얼통신 장비로부터 통신 인터페이스가 중요한 기술로 자리 잡고 있다. 본 논문에서는 공개된 범용프로토콜 Modbus 프로토콜을 기준으로 Modbus 프로토콜 규격인 ASCII, RTU, TCP의 인터페이스를 모두 활용 가능하고 I/O를 논리적으로 분리 가능한 소프트웨어를 설계한다[5].

II. Preliminaries

Fig. 1과 같이 기존에 절차지향으로 소프트웨어를 개발했을 경우, 시스템 구동에 필요한 I/O 장치를 해당 기능에 지속적으로 추가해야 한다. 프로그램 기능 추가, 수정 시 프로그램의 복잡도가 지속적으로 증가한다.

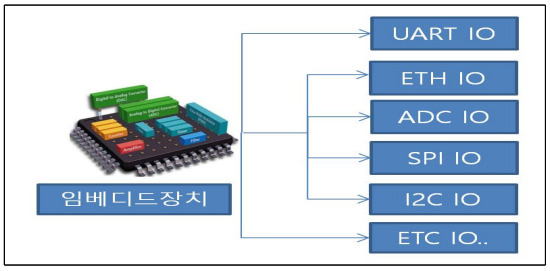

Fig. 1. Procedural Programming

Fig. 2는 본 논문에서 제시한 설계·구현 구조이다. 각 인스턴스에 개별 통신 I/O 함수를 등록하고 중개자 클래스를 통하여 출력을 요청하면, 등록되어진 통신 I/O 함수를 호출함으로서 출력을 수행한다. 즉 중개자 클래스에 등록되는 각 인스턴스는 개별 수행 로직을 가짐으로서 시스템 구현의 복잡성을 줄일 수 있다.

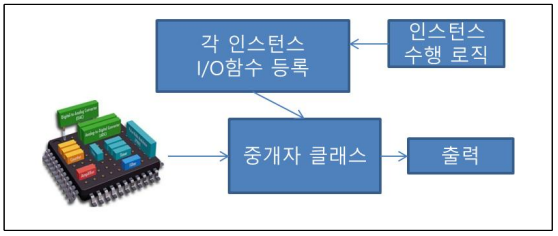

Fig. 2. Object-Oriented Programming

1. Modbus Protocol definition and LRC, CRC implementation

Modbus 프로토콜은 마스터-슬레이브 통신 방식을 사용하고 피어 투 피어 방식으로 통신한다. Fig. 3은 마스터와 슬레이브 간 요청 및 응답 사이클이다. 마스터는 슬레이브에 장비고유의 주소와 기능코드, 데이터, CRC를 전송하여 슬레이브로 부터 응답을 받고, 슬레이브는 내 자신의 고유주소와 기능코드, 데이터, CRC 값을 마스터에게 응답함으로써 이상 없이 통신이 완료 되었다는 것을 통지한다.

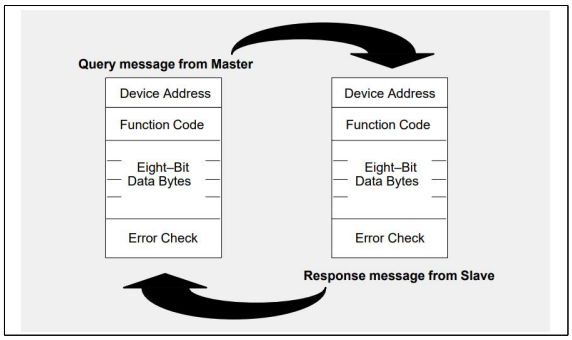

Fig. 3. Master-Slave Query-Response Cycle

Modbus 통신은 기본적으로 ASCII모드, RTU 모드를 지원하고 있다[6]. ASCII 모드의 사양은 아래 Table 1과 같다.

Table 1. Modbus ASCII Specification

Coding System:	Hexadecimal, ASCII characters 0-9, A-F
	One hexadecimal character contained in each
	ASCII character of the message
Bits per Byte:	1 start bit
	7 data bits, least significant bit sent first
	1 bit for even/odd parity; no bit for no parity
	1 stop bit if parity is used; 2 bits if no parity
Error Check Field:	Longitudinal Redundancy Check (LRC)

ASCII 모드는 Data bit는 7bit를 사용하고 있고, Parity 설정시 Even/Odd Parity를 사용할 수 있다. 데이터의 범위는 ASCII characters 0-9, A-F 이고 16진수로 변환시 0x30~0x3A, 0x41~0x46이 된다. 오류 검사 필드로 LRC가 사용되고 있다. 모든 필드의 값을 더한 후 2의 보수를 취하기만 하면 LRC 검사의 필드가 생성된다.

```
unsigned short LRC(char *puchMsg, int size)
{
    unsigned char uchLRC = 0;
    while(size--)
    {
        uchLRC += *puchMsg++;
    }
    return ((unsigned char))(-(uchLRC));
}
```

Fig. 4. LRC Generation Function

Fig. 4와 같이 LRC 생성 함수는 각 데이터 필드에 접근하여 각 더하기 연산 후에 2의 보수를 취하면 최종 결과인 LRC 연산이 완료되고 그 결과를 반환한다[7]. RTU 모드의 사양은 아래 Table 2와 같다.

Table 2. Modbus RTU Specification

Coding System:	8-bit binary, hexadecimal 0-9, A-F
	Two hexadecimal characters contained in each
	8-bit field of the message
Bits per Byte:	1 start bit
	8 data bits, least significant bit sent first
	1 bit for even/odd parity; no bit for no parity
	1 stop bit if parity is used; 2 bits if no parity
Error Check Field:	Cyclical Redundancy Check (CRC)

RTU모드는 8bit Binary를 사용하고 Parity 설정은 Even/Odd Parity를 사용한다. 데이터 범위는 16진수 0-9, A-F이고 16진수로 0x00~0xFF이다. 오류 검사 필드는 CRC가 사용되고, CRC생성은 아래와 같은 절차를 따른다. CRC생성 절차는 다음과 같다.

① 16 비트 레지스터를 FFFF 16진수 (모두 1로)로 로드한다. 이것을 CRC레지스터라고 한다.

② 16비트 CRC 레지스터의 하위 바이트가 포함된 메시지의 첫 번째 8비트 바이트를 제외하거나 CRC 레지스터에 결과를 입력한다.

③ CRC 레지스터를 오른쪽으로 1비트 이동(LSB방향으로)하고 MSB를 0으로 채운다. LSB를 추출하여 1인지 0인지 검사를 수행한다. LSB가 0인 경우, 3단계(다른 시프트)를 반복한다. LSB가 1인 경우, 다항식 값 A001 16진수(1010 0000 0000 0001)의 CRC레지스터 배타 연산 수행한다.

④ 8번의 비트이동 연산이 수행될 때까지 3단계와 4단계를 반복한다. 작업이 완료되면 전체 8비트 한 바이트가 처리된다.

⑤ 메시지의 다음 8비트 한 바이트에 대해 2~5단계를 반복하고 모든 바이트가 처리될 때까지 이 작업을 계속 수행한다.

⑥ CRC레지스터의 최종 결과는 CRC 값이다.

⑦ CRC를 프레임에 추가 할 시 아래 설명과 같이 CRC의 상한 바이트와 하한 바이트의 위치를 교환해야 한다.

RTU모드를 프로그램하면 Fig. 5와 같다.

```
unsigned short CRC16(char *puchMsg,int size)
{
    int register i;
    unsigned short crc, flag;
    crc = 0xffff;
    uint8_t usDataLen = size - 2;
    while (usDataLen--)
    {
        crc ^= *puchMsg++;
        for (i = 0; i<8; i++)
        {
            flag = crc & 0x0001;
            crc >>= 1;
            if (flag) crc ^= 0xA001;
        }
    }
    return crc;
}
```

Fig. 5. CRC Generation Function

Modbus ASCII의 프레임 구성은 그림 Fig. 6과 같다[4].

START	ADDRESS	FUNCTION	DATA	LRC CHECK	END
1 CHAR :	2 CHARS	2 CHARS	*n* CHARS	2 CHARS	2 CHARS CRLF

Fig. 6. ASCII Message Frame

Modbus ASCII 모드에서는 프레임 시작을 알리는 문자로 ':' 아스키 문자로 시작되고, 'CR''LF'사용하여 프레임의 끝을 알리고 프레임 전송이 종료된다. Fig. 4에서 ADDRESS는 슬레이브 기기 주소를 의미 하고 FUNCTION은 명령(기능코드) 'CR''LF'를 전송함으로서 전송 종료를 통지한다.

RTU 프레임은 ASCII 필드와 동일하며, 단지 시작과 끝을 알리는 문자가 없고, 시작 알림과 종료알림은 ASCII 와 다르게 START와 END에 시간차이를 주어 시작과 끝을 구분한다. Fig. 7의 시작 부분과 종료 부분의 T1-T2-T3-T4의 의미는 RTU 모드에서는 메시지 전송 시간으로 구분하고, 최소 3.5자의 문자 간격으로 시작과 끝이 구분된다.

START	ADDRESS	FUNCTION	DATA	CRC CHECK	END
T1–T2–T3–T4	8 BITS	8 BITS	*n* x 8 BITS	16 BITS	T1–T2–T3–T4

Fig. 7. RTU Message Frame

Modbus TCP/IP프로토콜은 기존의 ASCII, RTU 프로토콜과 비교하면 우선 오류 검사 프레임 필드가 존재 하지 않고 프레임 앞부분에 MBAP Header가 추가 되었다. MBAP Header의 필드는 Fig. 8과 같이 정의된다[6].

Fields	Length	Description -	Client	Server
Transaction Identifier	2 Bytes	Identification of a MODBUS Request / Response transaction.	Initialized by the client	Recopied by the server from the received request
Protocol Identifier	2 Bytes	0 = MODBUS protocol	Initialized by the client	Recopied by the server from the received request
Length	2 Bytes	Number of following bytes	Initialized by the client (request)	Initialized by the server (Response)
Unit Identifier	1 Byte	Identification of a remote slave connected on a serial line or on other buses.	Initialized by the client	Recopied by the server from the received request

Fig. 8. MBAP Header Fields

MBAP Header는 7Byte이고 각 필드별 역할은 다음과 같다.

① 트랜잭션 식별자 - 트랜잭션 페어링에 사용되며 모드버스 서버는 응답 시 요청의 트랜잭션 식별자를 응답으로 복사한다.

② 프로토콜 식별자 - 시스템 내부 멀티플렉싱에 사용됩니다. Modbus 프로토콜은 값 0으로 식별된다.

③ 길이 - 길이 필드는 장치 식별자 및 데이터 필드를 포함하여 다음 필드의 바이트 수이다.

④ 장치 식별자 - 이 필드는 시스템 내부 라우팅 용도로 사용된다. 일반적으로 이더넷 TCP-IP 네트워크와 모드버스 직렬 회선 사이의 게이트웨이를 통해 모드버스+또는 모드버스직렬 회선 슬레이브와 통신하는 데 사용된다. 이 필드는 요청에서 모드버스 클라이언트에 의해 설정되며 서버의 응답에서 동일한 값으로 반환된다[6].

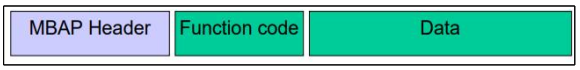

Fig. 9. Modbus TCP/IP Frame

Modbus ASCII, RTU, TCP프로토콜의 Function Code는 Fig. 10과 같이 정의된다[9].

Commonly used public function codes				
Code	Hex	Function	Type	
01	01	Read Coils	Single Bit Access	Data Access
02	02	Read Discrete Inputs		
05	05	Write Single Coil		
15	0F	Write Multiple Coils		
03	03	Read Holding Registers	16 bit Access	
04	04	Read Input Register		
06	06	Write Single Register		
16	10	Write Multiple Registers		
22	16	Mask Write Register		
23	17	Read/Write Multiple Registers		
24	18	Read FIFO queue		
20	14	Read File Record	File record access	
21	15	Write File Recore		
07	07	Read Exception Status	Diagnostics	
08	08	Diagnostic		
11	0B	Get Com event counter		
12	0C	Get Com Event Log		
17	11	Report Server ID		

Fig. 10. Modbus Function Codes

기능코드에는 3가지의 범주가 있다.

① 공공 기능 코드 - Modbus.org 커뮤니티에서 검증하고 공개적으로 문서화 하고 보장하는 코드번호 1~127까지있고 사용자 정의 코드는 제외한다.

② 사용자 정의 코드 - 65~72 및 100~110의 범위 내에서 사용자 정의 기능 코드를 지원한다.

③ 예약된 기능 코드 - 일부 회사에서 레거시 제품에 사용되며 공용으로 사용할 수 없다[6].

2. Class designed using builder pattern

Modbus ASCII, RTU, TCP프로토콜은 함수 기반의 언어로 코드 작성 시 복잡성이 증가한다. 특히 펌웨어 레벨에서는 함수 기반의 언어가 주로 사용되므로 통신I/O관련 함수를 수정 하거나 추가 해야되는 경우 소프트웨어의 복잡성이 증가한다. 따라서 함수의 네이밍 문제, 유지 보수의 어려움이 발생한다. 본 논문에서는 Modbus 프로토콜 ACCSII, RTU, TCP 3가지 기능을 활용하여 독립된 통신I/O 처리가 가능하고 유지 보수도 간편화하는 디자인 패턴 기반의 클래스를 설계하였다. 클래스 코드는 객체지향언어로 작성하였고 디자인패턴은 빌더 패턴을 참고하여 설계하였다.

Fig. 11에서 빌더 패턴 역할을 Modbus_director 클래스가 수행한다. Modbus_director 클래스는 Modbus_Ascii, Modbus_tcp, Modbus_rtu의 인스턴스를 매개변수로 받아 원하는 프로토콜 수행 로직을 스위칭 및 프로토콜전송 기능을 수행하는 클래스이다[8].

Fig. 11. Class Architecture

Modbus_Builder 클래스는 Modbus_director와 프로토콜 전송을 위한 인터페이스 함수를 제공하고 각 프로토콜을 정의 하는 클래스 Modbus_Ascii,Modbus_tcp,Modbus_rtu는 Modbus_Builder의 인터페이스 정의에 따라 구현하고 Modbus_director에 의해서 제어된다.

III. The Proposed Scheme

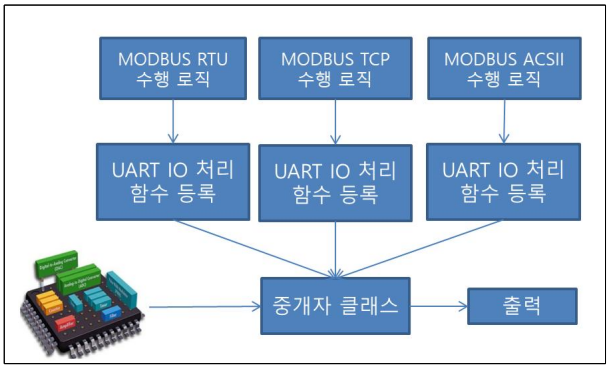

Fig. 12. System Implemented

Fig. 12는 본 논문의 최종 구성도이다. 각 인스턴스 마다 I/O처리 부분과 통신프로토콜을 생성하는 수행로직을 분리하였고 중개자 클래스트를 통하여 각 인스터스에서 구현 로직 및 등록되어진 I/O처리를 할 수 있다.

```
typedef unsigned short(*Error_Check)
(unsigned char *arg, int size);
class modbus_builder
{
    public:
    virtual void init(char* arr) {}
    virtual void m_adr(uint8_t m_adr){}
    virtual void fucntion_code
    (uint16_t func_code)=0;
    virtual void address(uint16_t adr)=0;
    virtual void value(uint16_t value)=0;
    virtual void check_sum
    (Error_Check fucntion)=0;
    virtual void send()=0;
}
```

Fig. 13. Modbus_Builder Interface Class

인터페이스 Builder클래스는 Fig. 13과 같이 가상함수로 정의된다. Modbus 프로토콜의 기본 프레임의 구조에 따라 반드시 필요로 되는 기초 함수 기기의 주소(M_adr), 기능 코드(Function_code), 메모리주소(Address), 값(Value), 오류 검사(Check_sum), 데이터전송(Send) 함수의 인터페이스를 제공한다.

```
typedef void (*fp)(unsigned char Data);
class modbus_rtu : public modbus_bulider
{
    private:
    unsigned char buffer[REQUEST_LEN];
    char *buffer_;
    char size_len;
    fp p;
    public:
    void init(char* arr);
    void m_adr(uint8_t m_adr);
    void fucntion_code(uint16_t func_code);
    void adress(uint16_t adr);
    void value(uint16_t value);
    void check_sum(Error_Check fucntion);
    void register_io_func(fp p);
    void send();
};
```

Fig. 14. Modbus_rtu Class

Fig. 14의 Modbus_rtu Class의 구조는 Modbus_Builder Class 인터페이스 코드이고 Tcp, ASCII의 생성 클래스도 동일하게 구현된다. typdef void (*fp)(unsigned char Data); 코드는 함수포인터의 원형이며 통신 I/O 처리는 바이트 단위로 데이터를 송신함으로 매개변수로 Byte처리 가능한 함수를 등록하게 정의 하였다.

```
class modbus_tcp : public modbus_bulider
{
    private:
    modbus_tcp_frame frame;
    unsigned char buffer[REQUEST_LEN];
    char *buffer_;
    char size_len;
    fp p;
    int to_little(int bit16);
    int to_big(int bit16);
    public:
    void init(char* arr);
    void m_adr(uint8_t m_adr);
    void fucntion_code(uint16_t func_code);
    void adress(uint16_t adr);
    void value(uint16_t value);
    void check_sum(Error_Check fucntion);
    void register_io_func(fp p);
    void send();
};
```

Fig. 15. Modbus_tcp Class

Fig. 15의 클래스는 TCP프레임을 처리하기 때문에 to_little, to_big 함수가 추가되어 시스템의 아키텍쳐에 따라 앤디안 변환이 가능토록 하였다.

```
class modbus_ascii : public modbus_bulider
{
    private:
    modbus_ascii_frame frame;
    unsigned char buffer[REQUEST_LEN];
    char *buffer_;
    char size_len;
    fp p;
    int to_little(int bit16);
    int to_big(int bit16);
    public:
    void init(char* arr);
    void m_adr(uint8_t m_adr);
    void fucntion_code(uint16_t func_code);
    void adress(uint16_t adr);
    void value(uint16_t value);
    void check_sum(Error_Check fucntion);
    void register_io_func(fp p);
    void send();
};
```

Fig. 16. Modbus_ascii Class

Fig. 16의 Register_io_func(fpp)는 Byte 디바이스 I/O 함수 포인터를 매개변수로 받고 Send()함수의 호출 시에 매개변수로 등록되어진 I/O 함수를 실행시킴으로서 데이터를 전송한다.

```
class modbus_bulider;
class modbus_director
{
    private:
    modbus_bulider *builder;
    public:
    explicit modbus_director
    (modbus_bulider *builder);
    void protocol_construct
    (uint8_t m_adr,uint16_t function_code,
    uint16_t adr, uint16_t value,
    Error_Check check_sum);
    void send_protocol();
};
```

Fig. 17. Modbus_director Class

Fig. 17의 Modbus_Director의 인스턴스 생성 시에 Builder 인스턴스의 주소를 전달받아 생성자 호출시 초기화되고, Protocol_construct 함수에서는 매개변수로 기기의 주소, 기능코드, 메모리주소, 값, 체크섬의 함수포인터의 주소 값을 받아 초기화된다. 최종적으로 Send_Protocol()함수를 호출하여 원하는 프로토콜을 전송한다.

```
modbus_rtu rtu = modbus_rtu();
rtu.init(nullptr);
rtu.register_io_func(UART_Putchar);
modbus_director dir = modbus_director(&rtu);
dir.protocol_construct(1,1, 20, 300, CRC16);
dir.send_protocol();
```

```
modbus_tcp tcp = modbus_tcp();
tcp.init(nullptr);
tcp.register_io_func(UART_Putchar);
modbus_director dir = modbus_director(&tcp);
dir.protocol_construct(1,1, 20, 300, nullptr);
dir.send_protocol();
```

```
modbus_ascii ascii = modbus_ascii();
ascii.init(nullptr);
ascii.register_io_func(UART_Putchar);
modbus_director dir = modbus_director(&ascii);
dir.protocol_construct(1,1, 20, 300, LRC);
dir.send_protocol();
```

Fig. 18. Using Modbus_rtu

Fig. 18와 같이 주 스레드에서 Modbus TCP, ASCII, RTU의 사용 예이다.

```
modbus_rtu rtu = modbus_rtu(); //인스턴스 생성
rtu.init(nullptr); //초기화
rtu.register_io_func(UART_Putchar);//IO함수포인터
modbus_director dir = modbus_director(&rtu);
//중계자 인스턴스
dir.protocol_construct(1,1, 20, 300, CRC16);
//국번1,기능코드1,주소20,데이터300
dir.send_protocol(); //프로토콜전송
```

Fig. 19. Using Instance In Main Thread

주 스레드에서 Fig. 19과 같이 전송 하고자 하는 프로토콜 객체를 인스턴스화 후 전송을 위한 시스템 I/O 함수의 주소를 Register_io_func의 인자로 전달하고, 실제 생성된 프로토콜은 인자로 전달한 I/O함수를 통하여 사용자가 지정한 I/O장치로 전송한다.

IV. Experiment

본 논문은 중소기업이 현장에서 직면하는 문제의 해결방안을 제시했으므로, 구현 방법이나 시스템 측면에서 제안된 방법의 비교 대상을 찾을 수 없었다. 또한 알고리즘 측면에서 비교할 만한 논문이 없었다. 따라서 제안된 방법의 구현 소스, 설정 방법 그리고 구현 결과에 중점을 두어 실험하고 평가한다.

본 논문의 실험 환경은 USB_TO_RS232통신 케이블을 사용하여 MCU(Atmel Avr128)장비와 통신 실험을 하였으며 소프트웨어 환경은 윈도우10 시리얼통신 모니터링 프로그램을 사용하여 실험하였다.

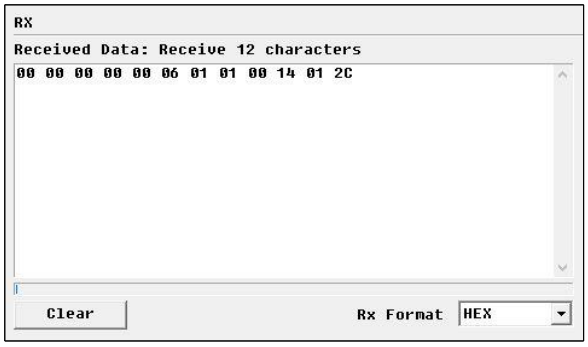

Fig. 20. Modbus_RTU Hex Frame.

Fig. 20은 Fig. 15의 첫 번째 Modbus rtu 수행 결과이다. 국번1, 기능코드1, 주소20, 데이터300으로 인자를 전송하여 Modbus_RTU Hex Frame을 전송한다.

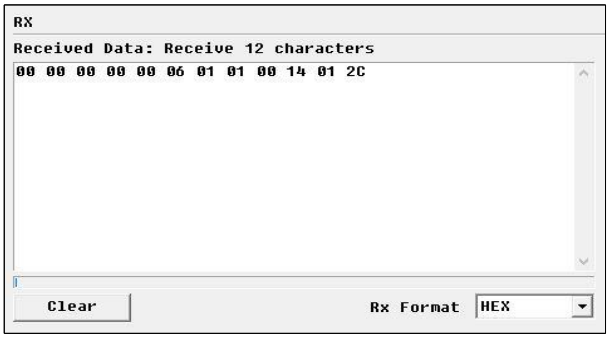

Fig. 21. Modbus_TCP Hex Frame.

Fig. 21은 Fig. 17의 두 번째 Modbus Tcp 수행 결과이다. 인자로 기능코드 1, 주소 20, 데이터 300을 전달한다.

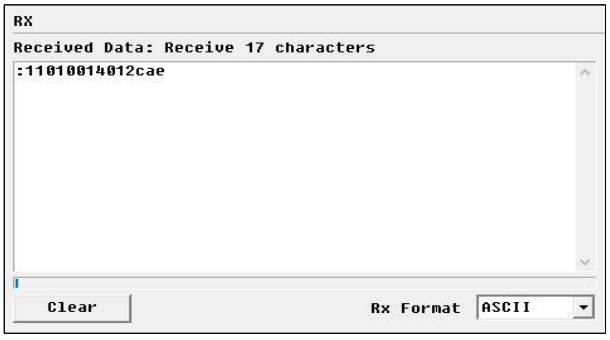

Fig. 22. Modbus_ASCII Frame.

Fig. 22는 Fig. 17의 세 번째 Modbus_ASCII 수행 결과이다. 인자로 국번 1, 기능코드 1, 주소 20, 데이터 300을 전달한다.

본 논문의 실험은 UART I/O장치를 활용하여 실험 하였고 중개자 클래스에 I/O함수를 각 인스턴스 I/O로 등록 하였다. 또한 인스턴의 일련의 작업(사용자 정의 프로토콜 생성)을 통하여 등록되어진 I/O장치를 통하여 송신을 확인 하였다.

V. Conclusions

스마트팩토리 시스템에서 통신 인터페이스는 매우 중요한 기술로서, 기존의 범용적인 OPC와 같은 통신 인터페이스 프로그램은 기존의 설비들과 이더넷만 연결되어 있고 여전히 지원되지 않는 프로토콜이 있다. 특히 이더넷기반의 서버이기 때문에 RS232 혹은 RS485와의 통신에는 제약이 있다. 본 논문의 통신 인터페이스는 시스템 I/O 함수를 프로토콜 생성자의 전송 I/O의 인자로 넘겨 RS232, RS485, Ethernet과 같은 물리적 부분은 I/O 함수를 통하여 사용자가 선택 할 수 있도록 분리 하였고 인자로 전달된 논리적 주소를 사용함으로서 I/O 함수와는 별개로 수행로직으로 구현된 모든 프로토콜은 생성과 전송을 가능하게 하였다. 본 논문은 I/O와 별개로 논리적으로 시스템을 설계하고 실험을 통하여 물리적 I/O를 동적으로 변동 가능하게 구현하였다. 이후 연구로는 고속의 인터페이스를 위하여 설비 장비의 메모리주소를 자동으로 탐색하고 탐색된 결과의 정보를 패킹하여 인터페이스를 쉽고 유연하게 개선할 수 있는 방법으로 연구를 확장해 나갈 계획이다.

REFERENCES

[1] Wikipedia, Fourth Industrial Revolution, https://ko.wikipedia.org/wiki/Fourth Industrial Revolution

[2] Wikipedia, Internet of Things, https://ko.wikipedia.org/wiki/Internet of Things

[3] Sangjin Jeong, Yoon-Young An, Hyunjoo Kang, Taehyoung Shim, Sung-Hei Kim, “An IoT Standards-Based Electrical Equipment Status Monitoring System Supporting Modbus/OCF Bridging,” Journal of Electrical Engineering & Technology, Vol. 69, No. 1, pp. 217-224, Jan 2020.

[4] Dong-Hwan Kim, Bo-Heon Kim, Jeong-Ho Song, Hwang-Rae Kim, “A Design of Modbus Communication Class for Multiple SCU Connections, “Journal of Korean Institute of Information Technology, Vol. 16, No. 2, pp. 67-73, Feb 2018, DOI 10.14801/jkiit.2018.16.2.67

[5] Sang-hee Eum, “A Programmable Protocol Data Conversion Algorithm for Industrial Machine Monitoring, “Journal of the Korea Institute of Information and Communication Engineering, Vol. 21, No. 11, pp. 2139-2144, Nov.2017

[6] Modicon Modbus Protocol Reference Guide, http://modbus.org/docs/PI_MBUS_300.pdf

[7] Microchip, http://ww1.microchip.com/downloads/en/DeviceDoc/doc2467.pdf

[8] Erich Gamma, Design Patterns , ADDISONWESLEY , 395, 2003

[9] Jeong-Ho Song, Bo-Hun Kim, Hwang-Rae Kim, “A Design of A Modbus Application Protocol for Multiple SCU Connections,“ Journal of the Korea Academia-Industrial cooperation Society, Vol. 19, No. 4, pp. 642-649, 2018 DOI 10.5762/KAIS.2018.19.4.642

[10] Modbus Application Protocol Manual V1_1b3: Available From: http://www.Modbus.org (accessed Aug, 15, 2017)

[11] Bo-Heon Kim, Jeong-Ho Song, Hwang-Rae Kim, “A Study on Enhancement of the MOD-BUS RTU Protocol for Multi-Device Connection”, Journal of KIIT. Vol. 16, No. 2, pp. 67-73, Feb. 28, 2018

Authors

Ki-Su Kim received the B.S., M.S. degrees in Computer Information Engineering from Kunsan University, Korea, in 2016, 2017, respectively. Ms. Kim is currently a Ph.d Course in the Department of Computer Information Engineering, Kunsan University. He is interested in Embedded Systems, PCB Design and Machine Learning, and Control System.

Jong-Chan Lee received the M.S. and Ph.D. degrees in computer science and engineering from Soongsil University, Korea, in 1996 and 2000 respectively. Dr. Lee was a senior member of engineering staff in Mobile Telecommunication Research Laboratory, Electronics and Telecommunications Research Institute (ETRI) from 2000 to 2005. Since 2005, he has worked in the Department of Computer Information Engineering, Kunsan National University as an professor. His current research interests are in the areas of dada analysis and deep learning