Timestamping over MODBUS TCP from Siemens S7

Joakim Wargh

Bachelor's thesis

Electrical Engineering

Vaasa 2013

BACHELOR'S THESIS

Author:	Joakim Wargh
Degree programme:	Electrical Engineering, Vaasa
Specialization:	Automation Technology
Supervisor:	Dag Björklund

Title: *Timestamping over MODBUS TCP from Siemens S7*

15.05.2013 32 pages 4 appendices

Abstract

This Bachelor's thesis work was done for ABB Power Generation Systems in Vaasa. I have been given the task of researching the possibilities of sending timestamp information to external systems from the PLC. Timestamping involves storing time information about when certain events have occurred. The aim of my thesis is to get a complete system that is able to create and store time information for certain events in milliseconds since 1/1/1970. In order to get an accurate representation of the time a 48 bits resolution is used. Communication over MODBUS TCP/IP requires the data that is sent to be stored in bytes or word format. The results of this thesis work are a working program for sending timestamps over MODBUS TCP and a manual on how to use and set up a working MODBUS TCP connection.

Language: English

Key words: timestamping, Modbus tcp/ip,

EXAMENSARBETE

Författare:	Joakim Wargh
Utbildningsprogram och ort:	Elektroteknik, Vasa
Fördjupning:	Automationsteknik
Handledare:	Dag Björklund

Titel: *Timestamping over MODBUS TCP from Siemens S7*

15.05.2013 32 sidor 4 bilagor

Abstrakt

Detta examensrbete gjordes åt ABB Power Generation Systems i Vasa. Jag hade fått till uppgift att forska om möjligheterna om att sända timestamping information till externa system från PLC. Timestamping innebär att man lagrar tidsinformation när vissa händelser har inträffat. Syftet med detta lärdomsprov är att få ett färdigt system som ska kunna skapa och lagra tidsinformation för vissa händelser i millisekunder sedan 1.1.1970. Överföringen med modbus tcp/ip kräver att data som skickas ska vara i byte eller word-format och därför används 48 bits upplösning på tidsinformationen som skickas. Detta lärdomsprov resulterade i ett fungerande funktionsblock som skapar tidsdata och skickar dessa over MODBUS TCP samt en manual över hur man använder detta funktionsblock och hur man öppnar en fungerande kommunikation.

Språk: Engelska

Nyckelord: timestamping, Modbus tcp/ip

Table of contents

1 Introduction ... 1
1.1 Target ... 1
2 ABB ... 2
2.1 ABB Finland ... 2
3 MODBUS ... 3
4 MODBUS application layer ... 5
4.1 MODBUS data model ... 6
4.2 MODBUS Function Codes ... 8
5 MODBUS TCP/IP ... 13
5.1 Messaging over TCP/IP ... 13
5.1.1 MBAP header ... 14
5.1.2 MODBUS request message ... 15
5.1.3 MODBUS response message ... 15
5.2 TCP connection management ... 15
6 Siemens S7 programming ... 17
6.1 IEC 61131-3 ... 17
6.2 SCL programming ... 18
6.3 FBD programming ... 19
7 Programming of the function block ... 20
7.1 Implementation specification ... 20
7.2 Timestamping ... 20
7.2.1 UNIX time ... 21
7.3 Conversion between Date and Time to UNIX time ... 21
7.3.1 Conversion to Unix time ... 22
7.3.2 Converting Unix time to array of bytes ... 25
7.4 MODBUS TCP/IP communication ... 26

7.4.1 Setting up a MODBUS TCP connection 27

7.4.2 Read data 28

8 Results 29

9 Discussion 30

10 References 31

APPENDICES

ABBREVIATIONS

ADU - Application Data Unit

HMI - Human Machine Interface

I/O - Input/Output

IP - Internet Protocol

MAC - Media Access Control

MB - MODBUS Protocol

MBAP - MODBUS Application Protocol

PDU - Protocol Data Unit

PLC - Programmable Logic Controller

TCP - Transmission Control Protocol

OSI - Open Systems Interconnection

Foreword

I would like to thank my supervisors, Mr. Dag Björklund at Novia University of Applied Sciences and Mr. Tomas Hultholm at ABB, for all the support and guidance they have offered me throughout this project.

1 Introduction

This Bachelor's thesis is made for ABB Oy Power Generation in Vaasa. This thesis work is a tool for ABB to send timestamp data to external systems and will be used in future projects.

1.1 Target

With an increasing number of different control systems working together in different processes it becomes more and more important to be able to compare alarm and event lists during fault searching. To be able to understand the reason for a fault it is very helpful to know the exact order in which events have occurred. The purpose of this thesis work is to investigate the possibility to send timestamp over MODBUS TCP/IP together with alarms/events to an external system.

This Bachelor's thesis will consist of the following steps:

- Investigation of the possibility to send timestamp over MODBUS TCP/IP to external systems
- Programming of a Function Block (FB) for Siemens S7 PLC that can be used to send timestamp over MODBUS TCP/IP to external systems
- Testing of the new Function Block
- Documentation and description of the new Function Block

2 ABB

By the merger of ASEA (Allmänna Svenska Elektriska Aktiebolaget) and BBC (Brown, Boveri & Cie) in 1988 a new company was founded, ABB (Asea Brown Boveri). ABB is one of the largest engineering companies specialized on power generation and power transportation with approximately 145 000 employees in over 100 countries. /1/

2.1 ABB Finland

ABB employs about 7000 employees in Finland with the largest activities in Pitäjänäki in Helsinki and Strömberg Park in Vaasa. The unit in Vaasa, to which this thesis work was made, manufactures control systems solutions for hydro, diesel and gas power plants. /2/

3 MODBUS

MODBUS has been the industry's serial communication standard since 1979. MODBUS, which is placed at level 7 of the OSI model, is an application layer messaging protocol. MODBUS offers communication between client/server on many different networks or on many different buses. /9/

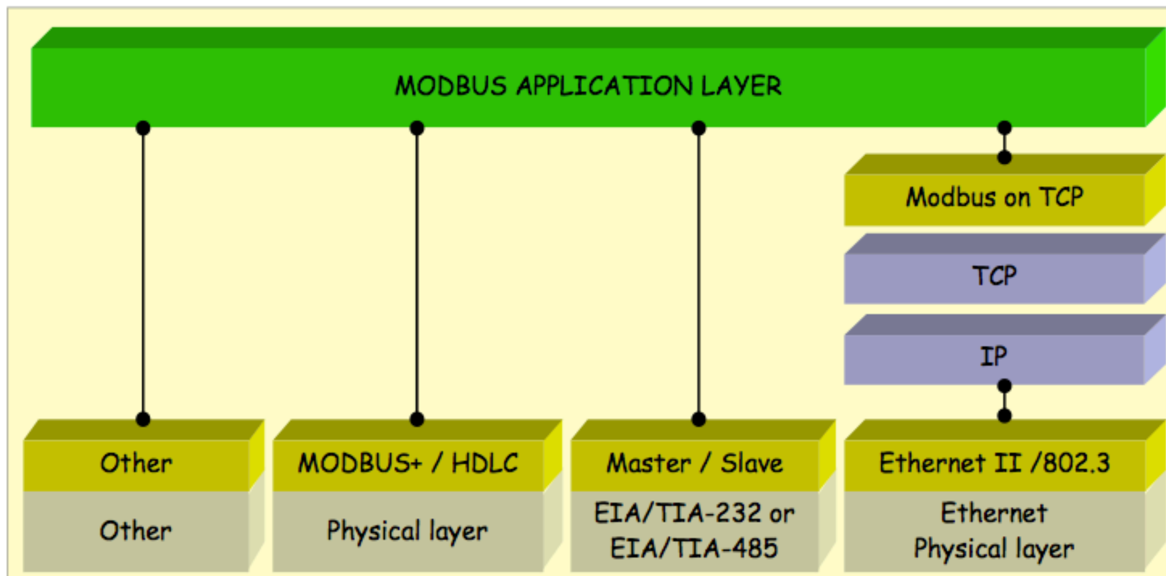


Figure 1 MODBUS communication stack /9/

MODBUS continues to grow and has been implemented to work with Ethernet and TCP/IP. Because the simplicity of MODBUS it is widely used around the world despite its old age. MODBUS is able to communicate on various buses or networks. Figure 1 shows the different physical layers that MODBUS has been implemented to work on. The figure also shows that all the different physical layers use the same application layer, which makes MODBUS easy to implement. It is currently implemented using:

- TCP/IP
- Asynchronous serial transmission (EIA-232-E, EIA-422, EIA-485. Fiber, radio, etc.)
- MODBUS plus

  /9/

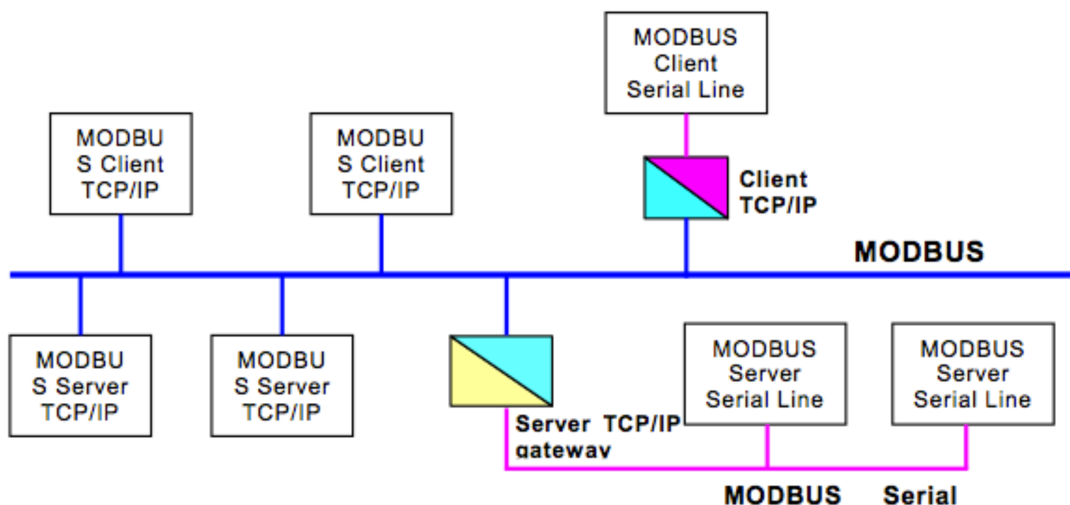


Figure 2 MODBUS communication architecture /8/

Figure 2 illustrates how MODBUS networks can be built using different physical layers, e.g. Ethernet, rs-232 etc. All MODBUS clients in the figure use the TCP/IP protocol (blue line), but three of the devices are connected to a MODBUS serial network (pink lines). In order for a MODBUS TCP/IP device to connect to a MODBUS serial device a gateway needs to translate the message into the correct format.

4 MODBUS application layer

The MODBUS protocol uses the polling principle, which means that the client asks every server that he is allowed to connect to if the servers have data to send to the client. When using the polling principle there will not be any collisions, because the client is the only device in a MODBUS network to initialize a connection. A MODBUS server cannot initialize a connection with a MODBUS client. A connection has to be initialized by a MODBUS client. The MODBUS master is also known as a MODBUS client and the MODBUS slave is known as a MODBUS server.

A MODBUS client is by definition in the MODBUS/TCP specification a very simple design. The three tasks a MODBUS client should be able to achieve are:

- To send an encoded request demanded by a user.
- To receive and analyze a response and send confirmation to the user application.
- To resend a package due to time out or send an error message to the user application.

A MODBUS server works as a service provider for a MODBUS client. The service provided can be e.g. grant access for a client to read object attributes, but it can also allow a client to set different settings/attributes in a server device (see figure 1). There are a few factors that affect how the server needs to be set up, e.g. if the user needs access to some advanced features in the device or if the user just needs access to the memory in the device. /8/

The MODBUS data frame is constructed of a Protocol Data Unit (PDU), which is a frame containing information like function codes and the actual data that the client needs from the server. The PDU is then included in an Application Data Unit (ADU). There are numerous function codes that have been predefined, such as the function code 03_{16} used for reading holding registers. Figure 3 shows a PDU frame and an ADU frame. /9/

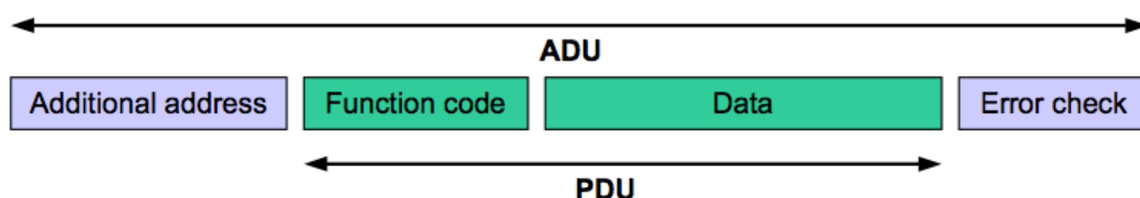


Figure 3 An example of a general MODBUS frame /9/

When a client initializes connection with a server the client sends a request message, which includes a function code so that the server knows what to do. The data sent to the server can contain information like how many bytes of data the slave has to send to the client, address information, etc.

If the message arrives at its destination without any errors, the server signals to the client that the transmission was a success (figure 4) by echoing the function code used. However if an error occurs the server uses an exception code (figure 10) to send to the client, which matches the error. /8/

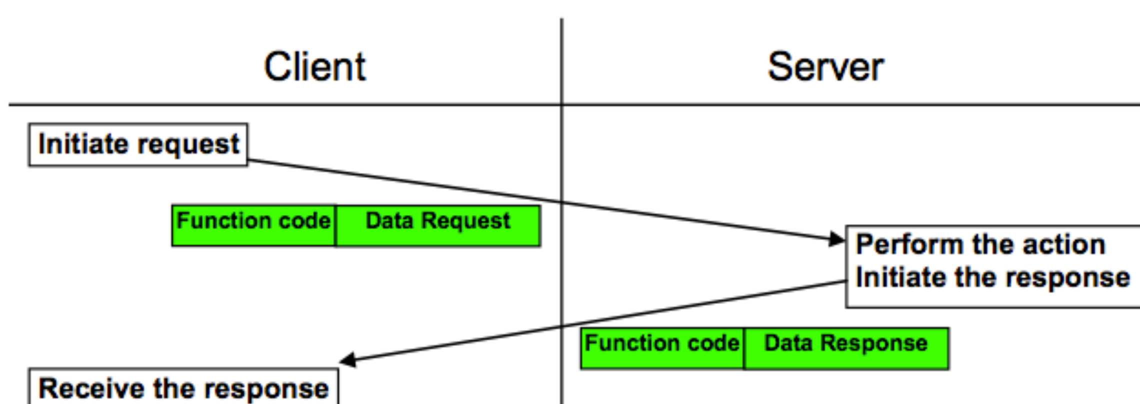


Figure 4 An example of a successful MODBUS transaction /9/

4.1 MODBUS data model

To represent data and addresses MODBUS uses "big-endian", which means that the most significant byte is sent first, followed by the less significant bytes.

Ex. 0x9876
The first byte sent is 0x98, followed by 0x76.

There are several ways to access data with MODBUS, bitwise or whole words at a time. Depending on what function codes you use, the reading or writing can vary from reading a whole register to reading just a bit.

Figure 8 shows the four tables used to store data in a MODBUS device. Both the input register and the input discrete register store data provided by the MODBUS device and are

therefore read-only registers. The only difference between the input registers are that the input discrete register stores the status of one bit (e.g. input from a switch) while the input register stores 16 bits of data from e.g. an analog signal. A user can both read and write to the coils register and the holding registers. The coils register is used whenever one needs to activate a single output (e.g. a relay) and the holding register can be used to store e.g. 16bits of numerical data like the input register. The difference between the input register and the holding register is that the user can only read data from the input register while data can be read and written to the holding register. It is possible and very common that all four tables overlap each other. The figure below illustrates a MODBUS device with separate data blocks. /9/

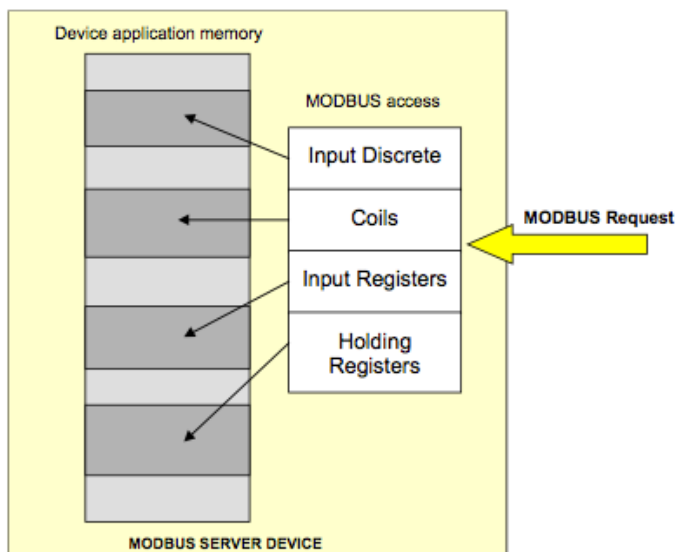


Figure 5 Example of a MODBUS device with four different data blocks /9/

To read and write to these different blocks (figure 8) one needs to use different function codes for all four blocks (appendix 1). E.g. if one wants to access the input discrete register, the function code (0x02) is used and if a user wants to write data to the coils register, the function codes (0x05, 0x0F) could be used. /8/

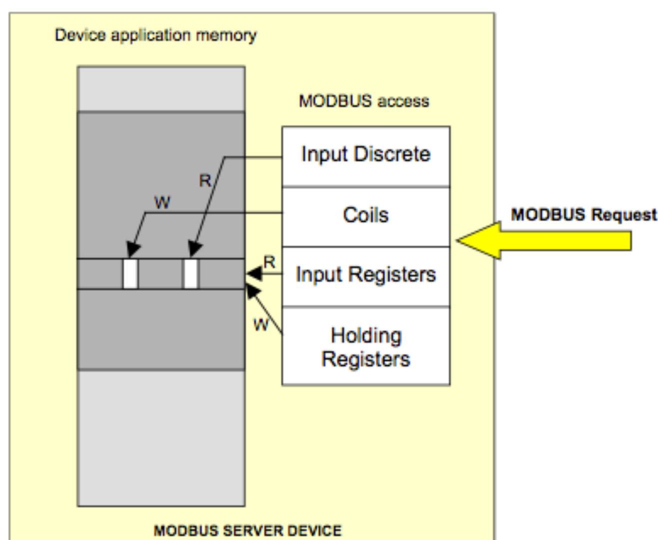


Figure 6 Example of a MODBUS device with one data blocks /9/

When the different tables overlap each other (figure 9), one can use the function read input register (0x04) to get the statuses for the whole register or one could use the read discrete input (0x02) to get the status of one bit in the register.

4.2 MODBUS Function Codes

Function codes in MODBUS are divided into three categories; public function codes, user defined function codes and reserved function codes (appendix 1). Public function codes are codes that are guaranteed to be unique and the MODBUS organization has to validate all public function codes. Additional to the validation and the uniqueness the function codes need to be available to the public.

Any function code in the range $1 \ldots 255_{10}$ is a valid function code, but the codes between 128_{10} and 255_{10} are reserved and used for exception responses. In cases where multiple actions are required a sub-function can be added to a function code. The function code requires a byte (8 bits) of memory in the PDU.

User defined codes are functions that anyone can implement with MODBUS, although these function codes are not supported in the MODBUS specification. User defined function codes are assigned two ranges, 65 to 72 and 100 to 110.

The reserved function codes are used by companies and for legacy reasons not available for public use. /9/

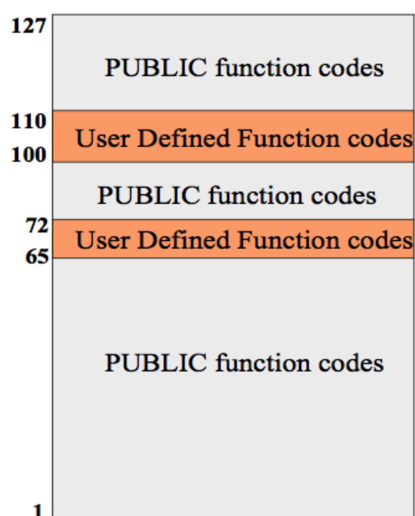


Figure 7 MODBUS function codes /9/

Below will follow an example of how a MODBUS PDU message is assembled when using the function code (0x01 read coils):

The read coil function is used to read coil statuses from a remote device. The request message sent to the server is assembled in the following way; the first byte is the function code, which in this case is 01_{16}, and the following two bytes are the starting address pointing to the first coil and the last four bytes sent in the request message represent the quantity of coils.

The response message is built in the same way as the request message. The first byte is the function code (in this case 0x01), the next bytes show us how many bytes to expect in the message and last but not least the statuses of the coils are sent. Figure 8 shows us the different tables presented in the previous sub-chapter. As shown in figure 8 the coils table can be both written to and read from.

Primary tables	Object type	Type of	Comments
Discretes Input	Single bit	Read-Only	This type of data can be provided by an I/O system.
Coils	Single bit	Read-Write	This type of data can be alterable by an application program.
Input Registers	16-bit word	Read-Only	This type of data can be provided by an I/O system
Holding Registers	16-bit word	Read-Write	This type of data can be alterable by an application program.

Figure 8 The four main tables in the MODBUS data model /9/

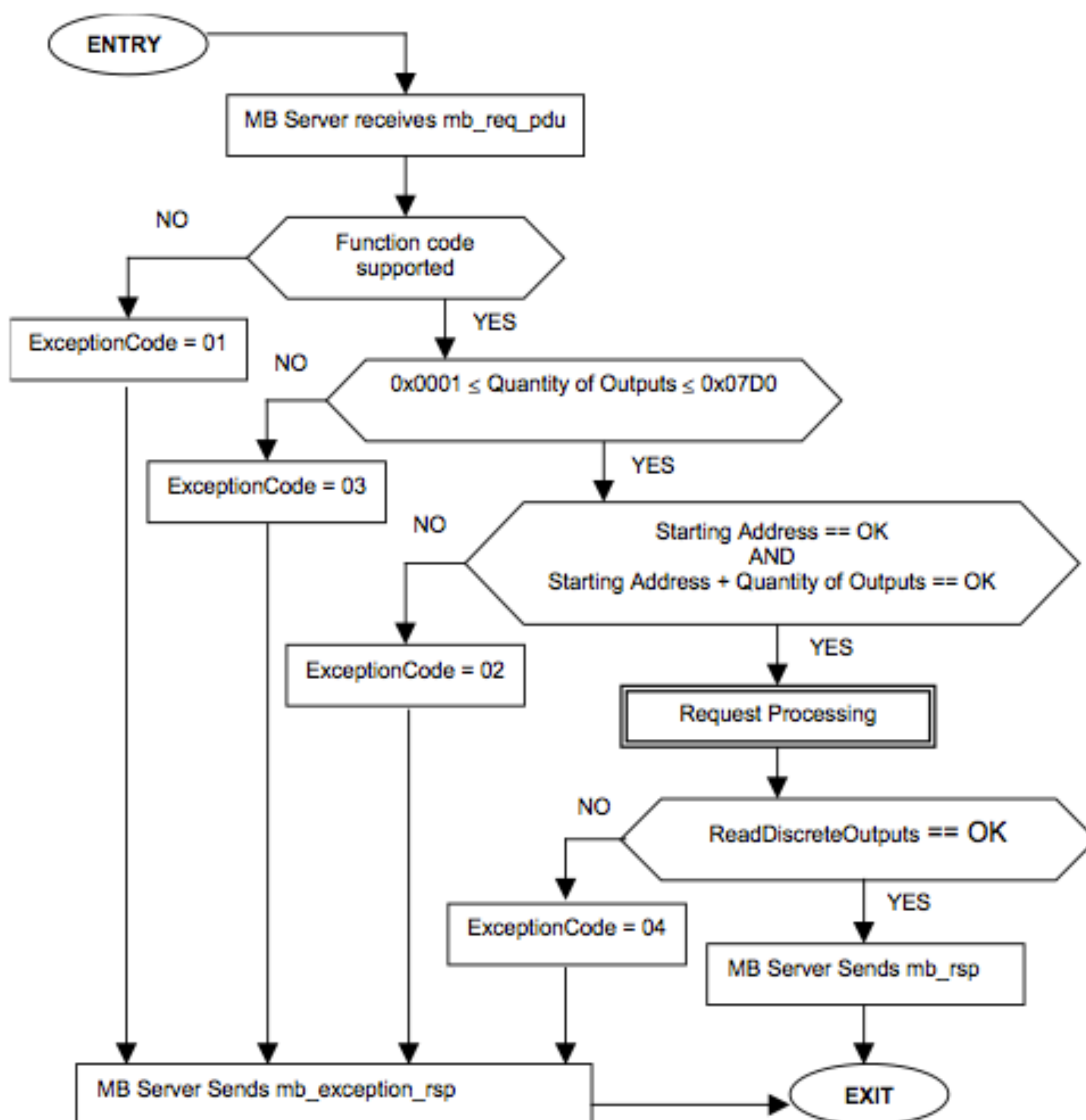


Figure 9 Read coil state diagram, MODBUS client /9/

Figure 9 shows a Read coil function (0x01). The server receives a request message from a MODBUS client. The server checks if the server supports the function code sent in the message. If the starting address is not a valid address in the server, the server sends an exception code. However, if the request passes every check that the server performs, a response message is sent to the client. The response message includes the function code sent by the client, in this example (0x01), followed by the requested statuses of the coils.

Exception Code	MODBUS name	Comments
01	Illegal Function Code	The function code is unknown by the server
02	Illegal Data Address	Dependant on the request
03	Illegal Data Value	Dependant on the request
04	Server Failure	The server failed during the execution
05	Acknowledge	The server accepted the service invocation but the service requires a relatively long time to execute. The server therefore returns only an acknowledgement of the service invocation receipt.
06	Server Busy	The server was unable to accept the MB Request PDU. The client application has the responsibility of deciding if and when to re-send the request.
0A	Gateway problem	Gateway paths not available.
0B	Gateway problem	The targeted device failed to respond. The gateway generates this exception

Figure 10 MODBUS exception codes /10/

If an error occurs in the communication between the server and client, or if errors occur when the server tries to execute the function code sent from the client, an exception code is sent to the client to inform about the error. Figure 10 shows the exception codes and in what type of situation the exception codes are used.

5 MODBUS TCP/IP

This chapter will explain which parameters and implementation are needed to send messages over MODBUS TCP/IP compared to “regular” MODBUS.

To get a clear view of how the communication between MODBUS devices works, a good knowledge of the TCP/IP protocol is necessary, as well as the knowledge of how to implement the MODBUS TCP/IP.

5.1 Messaging over TCP/IP

If the message sent over MODBUS TCP/IP is compared with MODBUS over serial line (see figure 1), the part that differs from “regular” MODBUS is the MBPA Header.

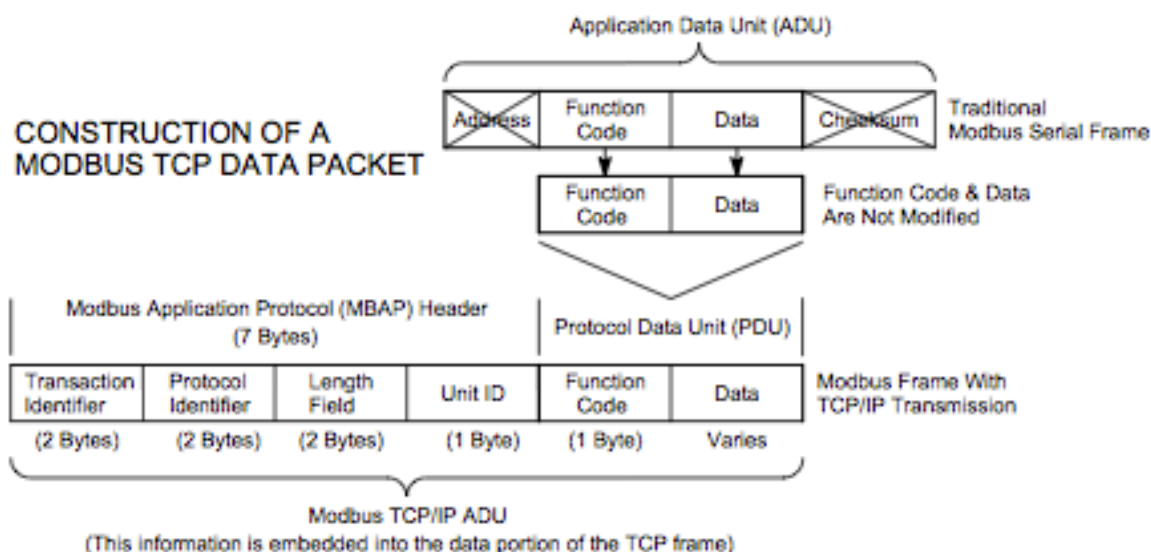


Figure 11 MODBUS data package over TCP/IP /6/

A unit identifier is used to reach the right device when sending messages over serial “traditional” MODBUS. For the receiver to know when and if the message was correctly received a byte count byte is added to this header. If the function code has a specified length the function code alone suffices. /9/

Additional length information is required if the message is split into several packets, and is also carried in the MBAP header. /9/

5.1.1 MBAP header

The following subchapter describes the usage of the MBAP header (figure 11) used in TCP/IP messaging over MODBUS. The MBAP header is 7 bytes long in a MODBUS TCP message.

- Transaction identifier
  The transaction identifier is used to identify future responses with this request. There are a few possibilities to implement this transaction identifier. One way is to use a counter that increases by one every time a request is sent.

- Unit identifier
  The unit identifier is only used to locate a MODBUS device on a serial-line, which is connected to a TCP/IP network through a gateway. The IP-address is used to locate the gateway and the unit identifier is used by the gateway to locate the right MODBUS device on the serial-line network.

  However, the unit identifier is useless if you use only a TCP/IP network where the IP-address is used to locate the correct MODBUS device.

- Length field
  The length field is used to indicate how many bytes there are in the message.

- Protocol identifier
  The protocol identifier should always be (0x0000) in MODBUS transactions. /6/

5.1.2 MODBUS request message

When you have sent the request and receive a response there are several aspects you have to take into consideration. As an example, if the response is missing a transaction identifier the response must be discarded.

If the transaction identifier points to a request sent earlier, you need to process the response message to know what type of confirmation to send to the user application. /8/

5.1.3 MODBUS response message

When the MODBUS server has finished processing the request it is time to start building the response message. There are two ways to respond to a request, either with the function code that was used in the request or with an exception function code.

A positive confirmation is sent to the user application if the function code in the response is the same as the request function code, and if the response has the correct format. A positive confirmation is also sent if the function code in the response is a MODBUS exception code.

A negative confirmation is sent to the user application if the function code received is different than the function code in the request message or if the format is incorrect.

The response PDU is constructed in the same way as the request: a unit identifier, length of the package, a protocol identifier (0x0000 for MODBUS) and a transaction identifier. /8/

5.2 TCP connection management

To successfully send and receive messages over TCP/IP a TCP connection must be established between a client and a server. The listening port number 502 is reserved for MODBUS TCP communication and must always be available for listening purposes. However, it is not mandatory to use port number 502 for listening. A user can specify other ports to be used for listening, but port 502 must be and remain reserved for MODBUS TCP (figure 12).

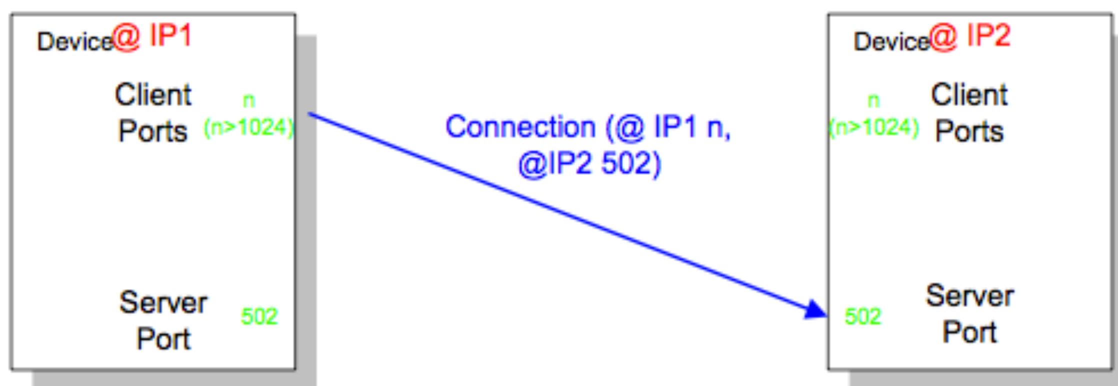


Figure 12 MODBUS TCP connection establishment /8/

There are two ways in which to manage a MODBUS TCP connection. One can let the user application module manage all connections, which is done for all connections that are set up between a server and a client. The other way is to let an automatic TCP connection manager be taken into use. A function has to be created, which closes the oldest unused connection in case the number of connections exceeds a maximum limit.

A connection is opened when a server receives the first message/package from a client.

An authentication of the IP addresses can be implemented in the access control module to prevent unauthorized IP addresses to connect to the device.

The access control module's function is to check every new connection to a list of IP addresses. If the IP address matches an IP address on this list, the connection is authorized and if not the connection is rejected. The user has to provide the IP address list to this module. /8/

6 Siemens S7 programming

This thesis work is planned to work with Siemens S7 300 and 400 series. Thus, the work and the program have to be programmed in a way that works with both series. This was not a problem because both series use the same programming languages. Both SCL programming and FBD programming have been used in this thesis work.

6.1 IEC 61131-3

The IEC 61131 is an open standard for programmable logic controllers, it consist of 8 parts. The IEC 61131 standard deals with e.g. testing of a PLC, communication standards and programming languages. IEC 61131-3 is the third part of this standard and this part deals with the programming languages used by a PLC.

The IEC 61131-3 defines 5 programming language standards. Two of the programming languages are text-based languages and the rest are graphical programming languages. /5/

These programming languages are:

- Graphical
  - LD (Ladder diagram)
  - FBD (Function block diagram)
  - SFC (Sequential function chart)
- Text-based
  - ST (Structured text)
  - IL (Instruction list)

6.2 SCL programming

SCL (Structured Control Language) is a high-level programming language used to program various blocks with Siemens logic controllers, and is defined as a ST (Structured text) language. The SCL language meets the requirements set for the ST language in the 61131-3 standard. The SCL language is similar to PASCAL. SCL programming with SIMATIC S7 is made very easy and, because the SCL language supports many block concepts it is easy to create different and very complex functions with STEP 7 and SLC.

The SCL programming environment consists of three parts; an editor, a batch compiler and a debugger. The syntax that the editor uses is based on PASCAL, which has some similarities with C, and is therefore a very easy and powerful programming language. The batch compiler generating a MC7 (machine code) from the code created in SCL is executable in most STEP 7 300/400 PLC:s (newer than 314). The third and last part of the SCL language is a debugger used for error searching in real-time.

One of the features that make programming easy with SCL is premade templates, e.g. an IF-statement that only has to be filled in. Siemens has a lot of preprogrammed functions that are ready to use, e.g. conversation functions and system functions like the READ_CLK function, which reads the PLC system clock and saves the data in a DT variable (DateAndTime). The READ_CLK function has been used in this thesis work. /12/

6.3 FBD programming

The FBD language is a graphical programming language. The basic idea with FBD programming is drag and drop, e.g. a function or an operation (+,-,*,/) is selected from a list and dragged to the interface.

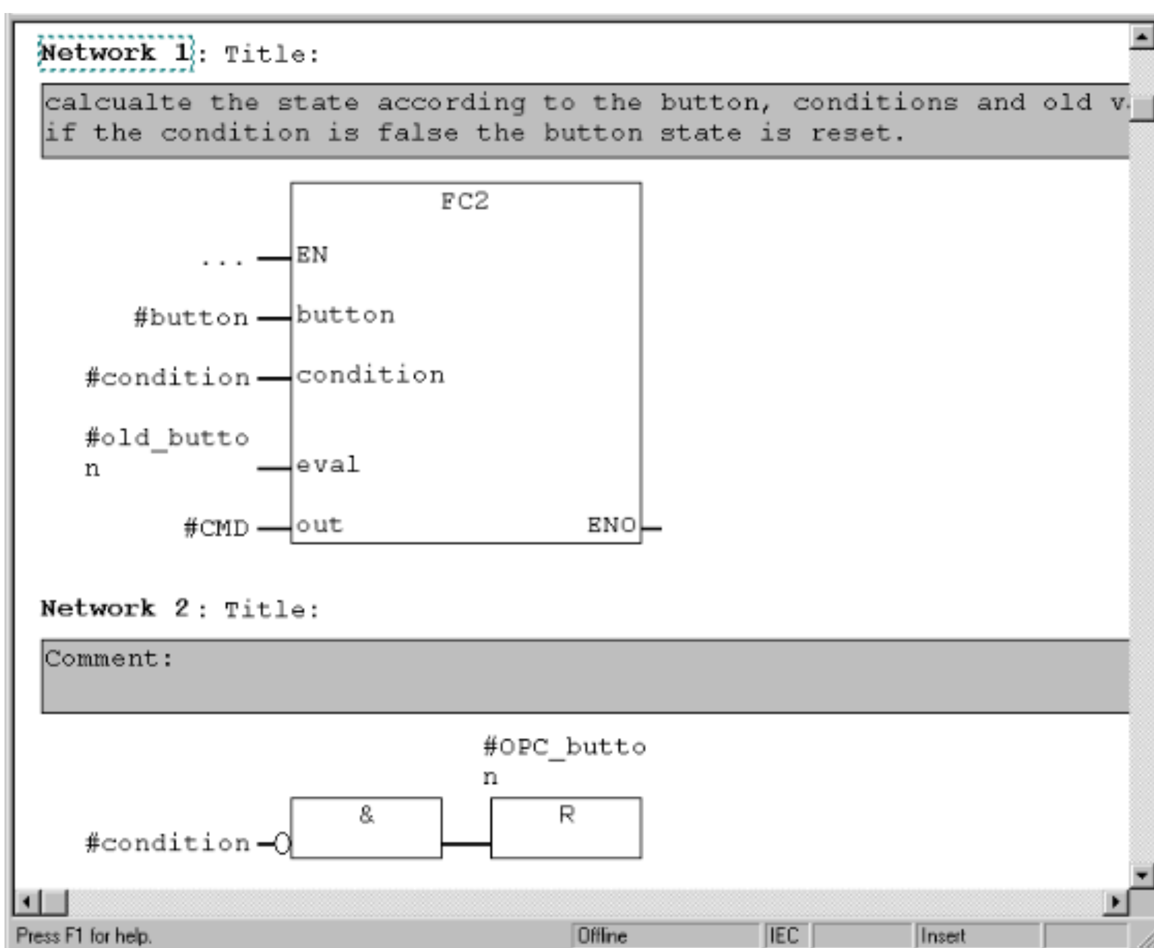


Figure 13 Picture of FBD interface and function blocks /10/

FBD offers a very simple way to connect multiple blocks. Even if the program is large and complex it is very easy to read and understand the program due to the simple and clear interface SIMATIC offers.

7 Programming of the function block

Two programming languages were chosen to implement the timestamping over MODBUS. The first language was SCL because it is an easy and powerful way to do the conversation from date and time to unix time. The second language was FBD and it was chosen because it offers a simple way to connect multiple function and function blocks. The SCL programming language has only been used in the conversation block and FBD in the rest of the program.

7.1 Implementation specification

The task of this thesis work was to program a function block in S7 that converts and sends timestamped data to a remote MODBUS device. The following steps are needed to implement the sending of timestamps over MODBUS TCP: a conversion from the Siemens DT variable to an array of bytes and setting up a connection to a remote MODBUS device.

The remote client, which is the overlying system, will read the timestamps from a MODBUS server using the function code read holding registers (0x03). The timestamps will be saved in the client's holding register together with the statuses of the timestamped data.

The timestamps will be in milliseconds to get an accurate representation on when an event has occurred.

7.2 Timestamping

This thesis work uses an existing block to create the actual timestamps. However, the timestamps are saved in a DT (Date and Time) variable. The block used is called a DEAMON (Appendix 2). This block has 32 inputs and whenever one of these inputs changes the DEAMON will timestamp that input. The timestamps are stored in an array of DT:s. The DEAMON gets the actual time and date data from the TS_DEALER, which reads the system clock.

7.2.1 UNIX time

In Unix time the definition of time is seconds since 1.1.1970 (UTC). Leap seconds are not accounted for. Unix time defines that one day is exactly 86400 seconds long. E.g. 1000 days after Unix epoch (1970-1-1T00:00:00Z ISO 8601) is represented as 86400000.

The format DI (double integer) uses 32 bits to store its data. This is sufficient to store Unix time in seconds, but at 03:14:07 UTC 2038-01-19 the DI format will overflow (called 2038 problem). /14/

There is no way to store integers larger than 2^{32} (approx. 4,3 mill) with SIMATIC, which will be a problem when one wants to use milliseconds since Unix epoch. The amount of bits needed to store Unix time in milliseconds is a minimum of 41 bits.

The timestamp will be stored in 6 bytes, which offer 48 bits of storage. This is sufficient for a very long time (approximately 8930 years). One byte is able to store 8 bits.

7.3 Conversion between Date and Time to UNIX time

The DT format is saved in 8 bytes in Step7. The first byte represents the current year (the two last numbers), the second byte represents month and the third byte represents day. (example: 13-03-04). The following 3 bytes represent hours, minutes and seconds. The two last bytes represent milliseconds and weekday (0 = Sunday…7 = Saturday).

However, the DT format is not supported by MODBUS, which is why the DT format needs to be converted. After some discussion and thinking about different formats and which format is most suitable for transporting timestamp information over MODBUS, it was decided that the format used in this thesis should be millisecond since 1.1.1970. This format is already in use by the corporation Wärtsilä and was therefore considered as the best solution.

7.3.1 Conversion to Unix time

Because Step 7 stores the DT format as 8 bytes it is necessary to convert DT to double integers (32 bits), before any calculations are done. The conversion to integers is done to make the different calculations easier. If one uses an array format to store the DT one can separate the individual bytes.

The variables `DaT` and `DateAndTime` contain exactly the same data, the difference is that the `DaT` variable points to the same memory address. Because the `DaT` variable is an array of 8 bytes one can access the individual bytes in the `DateAndTime` variable. The `Temp_Year` variable stores the last two numbers of the current year e.g. 2013 is stored as 13. The conversion between bytes and integer cannot be done directly, it needs to be converted first to the word format before it can be converted to an integer.

```
DateAndTime                                                  :DT;
DaT AT DateAndTime :ARRAY[0..7] OF BYTE;

Temp_Year[A]:=INT_TO_DINT(BCD_TO_INT(WORD_TO_BCD(BYTE_TO_WORD
             (DaT[0]))));
```

The rest of the bytes in `DateAndTime` are converted the same way as the `Temp_Year[]` variable.

```
Year := 1972;

WHILE Year < (2000 + Temp_Year[A]) DO
     IF (((Year MOD 4) = 0) AND NOT ((Year MOD 100) = 0)) OR
     ((Year MOD 400) = 0) THEN
     LeapYear := LeapYear + 1;
     END_IF;
Year := Year + 4;
END_WHILE;
```

It is important to take leap years into account. The code above will calculate the amount of leap years since 1970. If leap years were not considered in the Unix time conversion, the end result of the conversion would be missing 11 days or 950 400 seconds (in 2013). The leap year calculations are easily done by using the modulo function. If the result of `MOD 4` is zero and `MOD 100` is not zero, the `Year` th year is a leap year or, if the result of `MOD 400` is zero, the `Year` th year is also a leap year and the variable `LeapYear` will be incremented by one. The variable `Year` will start at 1972 and will loop until `Year` is

greater than the current year. This leap year calculation will result in an integer that holds the amount of leap years since 1970 and it will be used in the Unix time calculations.

The code below calculates the Unix time. Depending on if the current year is a leap year or not, there are two ways of calculating the Unix time. The first if-statement checks if the current year is a leap year and the second if-statement checks if the current month is March to December. `TempH` and `TempL`, which are integers and are able to store 32 bits of data, are temporary variables used to store the Unix time. `TempH` stores the amount of seconds from 1.1.1970 until today. The `TempL` variable stores the amount of milliseconds from midnight of the current day.

```
IF (((Year MOD 4) = 0) AND NOT ((Year MOD 100) = 0)) OR
((Year MOD 400) = 0) THEN
     IF (Temp_Month[A] > 2) THEN
          TempH := (((((Temp_Year[A]+2000)-1970)*365)
          +LeapYear)+((367*Temp_Month[A]-362)/12
          +Temp_Day[A] - 2))*86400;
     ELSE
          TempH := (((((Temp_Year[A]+2000)-1970)*365)
          +LeapYear)+((367*Temp_Month[A]-362)/12
          +Temp_Day[A]))*86400;
     END_IF;
ELSE
     IF (Temp_Month[A] > 2) THEN
          TempH := (((((Temp_Year[A]+2000)-1970)*365)
          +LeapYear)+((367*Temp_Month[A]-362)/12
          +Temp_Day[A] - 3))*86400;
     ELSE
          TempH := (((((Temp_Year[A]+2000)-1970)*365)
          +LeapYear)+((367*Temp_Month[A]-362)/12
          +Temp_Day[A]))*86400;
     END_IF;
END_IF;

TempL := (((Temp_hour[A]*3600)-(GMT*3600))
          +((Temp_Minute[A])*60)+(Temp_Second[A]*1000)
          +Temp_MSecond[A]; /3/
```

The following step shows how the Unix Time is calculated and it makes the code easier to understand and easier to read.

The amount of days since 1970 is always calculated in the following way:

Take the current year and add 2000 to it, subtract 1970 from the result to get the amount of completed years between 1970 and the current year. This is then multiplied by 365 to get

the amount of days. It should be noted that this will result in an answer that is missing the amount of leap years. The `leapyear` variable, which contains the amount of leap years, is added to the results and is then saved in the `TempH` variable.

Calculating the amount of days since the beginning of the current year:

The following formula is used to calculate the amount of days since the beginning of the current year.

Start with 367 multiplied by the current month, subtract 362 from the result of the multiplication. Then divide the result with 12 and add the current date. If the current year is a leap year and if the current month is larger than 2 one will need to subtract two days from the result, one day to compensate for the leap year and one day to exclude the current day. The current day is excluded from the result because it is not a completed day and it would therefore contribute to make the Unix time show the wrong time.

To clarify this an example is given below, let us say that the date is 18 April 2013.

2013 is not a leap year.

$$\frac{367 \cdot 4 - 362}{12} + 18 - 2 = 108{,}16$$

Because the result is saved in `TempH`, which is an integer, the decimal part will be left out.

Calculation of seconds since 1.1.1970:

The `TempH` variable now contains the amount of days between 1.1.1970 and the current day. It is easiest to convert the days into seconds by multiplying the `TempH` variable by 86400, which is the amount of seconds in one day.

Calculating the amount of milliseconds between midnight and now:

The variable `TempL` will be used to store the Time part of `DateAndTime` variable. It is very easy to calculate the amount of milliseconds since midnight. One needs to multiply the `Temp_Hour` variable by 3 600 000, the `Temp_Minute` variable by 60 000, the `Temp_Second` variable by 1000 and last but not least add the `Temp_MSecond` variable. Because the Unix time is UTC +0 one will need to subtract the amount of hours that a country is ahead or after the UTC +0 (Finland is UTC+2).

7.3.2 Converting Unix time to array of bytes

Now we have two variables that together contain the Unix Time.

This code below will take the Unix time calculated in chapter 7.3.1 and convert it to an array of 6 bytes. This is achieved by right shifting the data different amount of times.

The SHR function is used to right shift any word or byte N bits.

An example is given below of how to make the SHR function understandable, data = 0x1234 and N = 8.

SHR(IN:= 0x1234, N:=8) would result in 0x0012.

The value of N decides how many steps the data will be shifted. An N value of 8 will shift the data a whole byte to the right and an N value of 16 will shift the data two bytes to the right.

```
Byte[A+0]:=DWORD_TO_BYTE(SHR(IN:=DINT_TO_DWORD(DWORD_TO_DINT
           (SHR(IN:=DINT_TO_DWORD(TempH),N:=16))*1000,N:=24));
Byte[A+1]:=DWORD_TO_BYTE(SHR(IN:=DINT_TO_DWORD(DWORD_TO_DINT
           (SHR(IN:=DINT_TO_DWORD(TempH),N:=16))*1000,N:=16));
Byte[A+2]:=DWORD_TO_BYTE(SHR(IN:=DINT_TO_DWORD(TempH
           *1000+TempL),N:=24));
Byte[A+3]:=DWORD_TO_BYTE(SHR(IN:=DINT_TO_DWORD(TempH
           *1000+TempL),N:=16));
Byte[A+4]:=DWORD_TO_BYTE(SHR(IN:=DINT_TO_DWORD(TempH
           *1000+TempL),N:=8));
Byte[A+5]:=DWORD_TO_BYTE(DINT_TO_DWORD(TempH*1000+TempL));
```

A DWORD in Step 7 can only store 32 bits of data, so therefore the two most significant bytes must be shifted two times in order to get access to all data. The variable TempH variable needs to be multiplied by 1000 to convert from seconds to milliseconds. After one has multiplied TempH with 1000 one can add the TempL variable, as shown in the code above.

This thesis uses an existing function that creates timestamp information and saves this information in an array of DT. The (DT – Unix time) converting block then converts the array of DT timestamps and saves the information in an array of bytes.

7.4 MODBUS TCP/IP communication

The remote MODBUS client tries to connect to a Siemens S7 PLC, which acts like a server. The MODBUS server holds all timestamp data and the statuses of the timestamped bits. The PLC (server) has an IP address that the MODBUS client connects to. The MODBUS client sends a request message to the MODBUS server asking for the timestamped bits. The MODBUS server's task is to collect the timestamp data and send it to the remote client.

The user needs to give the PLC, the MODBUS server, an id in order for the remote MODBUS client to be able to locate the MODBUS server. The user also needs to specify the address that the timestamped data are stored at.

The following blocks are used to open a TCP connection with a remote client:

- MBTCP SERVER

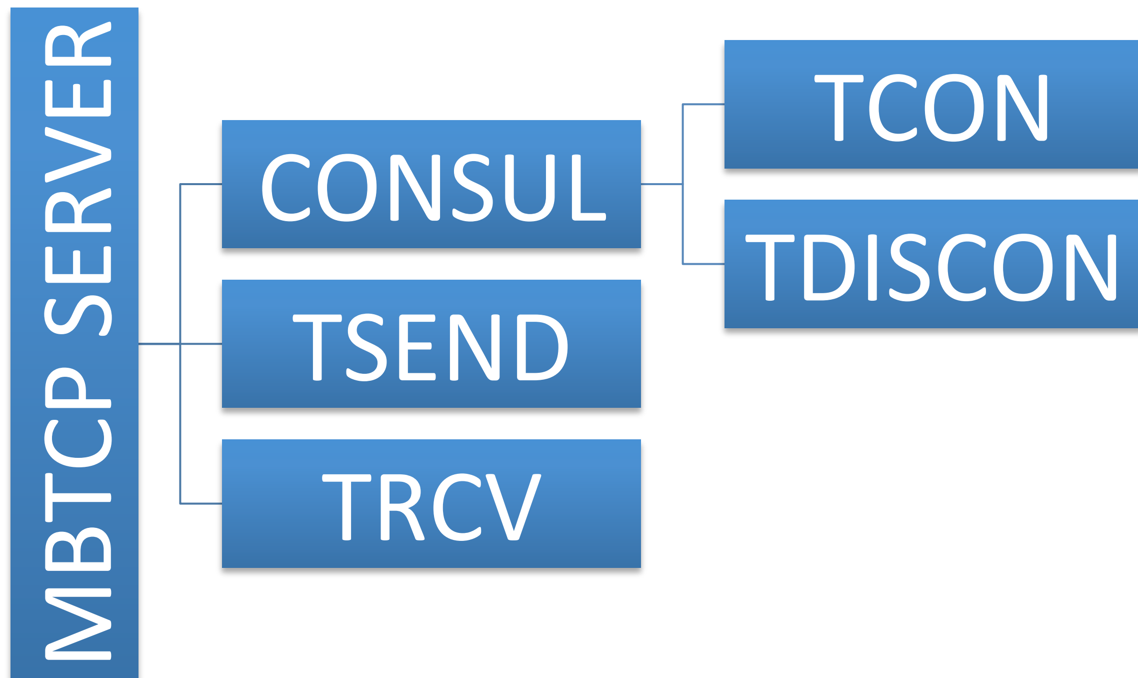
  This block allows the client to connect to the MODBUS server. In order to be able to open the connection it needs to use the block listed below. The blocks are a part of the MBTCP SERVER block.

- TSEND

  This block is used to send data to a MODBUS device

- TRCV
  This block is used when receiving data from another MODBUS device.
  - CONSUL
    Assigns a remote port number to the MORDBUS server and allows the MODBUS client to connect to the MODBUS server
  - TCON
    This block is used to open a connection with a remote device.
  - TDISCON
    This block is used to shut down an active connection.

7.4.1 Setting up a MODBUS TCP connection



The MBTCP SERVER takes the ID and the start address the user has specified and if the remote MODBUS client wants to connect to the server, the MBTCP SERVER opens the connection.

The CONSUL block assigns a remote port number to the server, by default 502.

In addition to these settings there are many different settings one can apply to the TCON block. One can choose if the TCP protocol or ISO on TCP etc. is used. One needs to specify an interface through which the S7 can communicate with a remote device. The length of the parameters for the local and remote endpoints have to be specified.

7.4.2 Read data

When the CONSUL block has finished opening the connection with a remote client device the client is able to start reading data. The MODBUS client sends a request to the MODBUS server using the function code (0x03) to read the holding registers. The MODBUS server stores the timestamped data in the holding registers.

When the request has been processed and verified the MBTCP SERVER uses the TSEND block to send the data to the remote MODBUS client. The MODBUS server then sends back the response message.

If the request sent to the remote device fails due to some error in the connection, the MBTCP SERVER will receive an exception code and notify the error to the user.

To validate the message sent from the remote device one can compare the transaction identifier sent in the request message and the transaction identifier in the received message, and if they do not match the message is discarded.

8 Results

The first task I was given was to research if it is even possible to send timestamp data over MODBUS TCP. After some research I came to the conclusion that it is possible to send timestamp data. The only modification to the timestamp data before it could be sent was that it had to be converted into an array of bytes.

The biggest task was to figure out how to convert the timestamps into an array of bytes. After some experimenting and testing a working program was created and the task of setting up a connection between two devices remained. This part of my thesis work was relatively painless because of the premade blocks that are used for MODBUS TCP messaging at ABB.

The results of this thesis work are a working program for sending timestamp data from a Siemens S7-300/400 and a manual for how to set up a working connection between a remote device and how to use the function block that I created.

There is a possibility to expand this program to use the function code (0x65). Wärtsilä uses the function code to read timestamps from a device. In order to get this function code to work one will need to modify the MBTCP SERVER block to support this function code. Today the MBTCP SERVER block supports these function codes: 0x03, 0x04, 0x06 and 0x10.

9 Discussion

In the beginning a great amount of time was spent familiarizing myself with SCL programming, as I had never used this programming language before. Even if I was new to the programming language it did not take long before I saw some similarities with C programming. Because I was familiar with C programming the main issue was to get an understanding of how the syntax of SCL programming is.

After I got familiarized with SCL programming I needed to get an understanding of how I wanted the program to work. After some trial and error I got a clear picture of how the program should work and I started to create the program. The program had to be remade a couple of times because of some minor bugs and faults. Finally I got a working program and was able to start testing it.

During this thesis work I got a better understanding of how programming with S7 works and my programming skills increased a lot.

10 References

/1/ ABB

http://www.abb.com

(Read 17.3.2013)

/2/ ABB Finland

http://www.abb.fi/

(Read 17.3.2013)

/3/ Calculating day of year

http://www.dispersiondesign.com/articles/time/calculating_day_of_year

(Read 20.3.2013)

/4/ Internet protocol suite

http://en.wikipedia.org/wiki/Internet_protocol_suite

(Read 26.2.2013)

/5/ IEC 61131-3

http://en.wikipedia.org/wiki/IEC_61131-3

(Read 29.4.2013)

/6/ Introduction to MODBUS TCP/IP (2005)

http://www.dee.hcmut.edu.vn/vn/ptn/sch/download/Network_Architecture/intro_modbusTCP.pdf

(Read 20.3.2013)

/7/ MODBUS

http://en.wikipedia.org/wiki/Modbus

(Read 25.2.2013)

/8/ MODBUS TCP/IP (2006)

http://modbus.org/docs/Modbus_Messaging_Implementation_Guide_V1_0b.pdf

(Downloaded 2.2.2013)

/9/ MODBUS protocol specification (2012)

http://modbus.org/docs/Modbus_Application_Protocol_V1_1b3.pdf

(Downloaded 2.2.2013)

/10/ Picture of FBD interface and function blocks

http://en-dep.web.cern.ch/en-dep/Groups/ICE/Services/PLC/Images/userpro.gif

(Downloaded 4.3.2013)

/11/ SIEMENS FBD programming manual (2010)

http://cache.automation.siemens.com/dnl/jg/jg0NDM4OQAA_45522487_HB/s7fup__b.pdf

(Downloaded 4.3.2013)

/12/ SIEMENS SCL programming manual (2005)

http://cache.automation.siemens.com/dnl/zMxOTcwMwAA_5581793_HB/SCL_e.pdf

(Downloaded 26.2.2013)

/13/ Timestamp over MODBUS TCP/IP (2007)

ABB Internal document

(Read 1.2.2013)

/14/ UNIX time

http://en.wikipedia.org/wiki/Unix_time

(Read 4.3.2013)

APPENDIX 1

				Function Codes		
				code	*Sub code*	*(hex)*
Data Access	**Bit access**	Physical Discrete Inputs	Read Discrete Inputs	**02**		**02**
		Internal Bits Or Physical coils	Read Coils	**01**		**01**
			Write Single Coil	**05**		**05**
			Write Multiple Coils	**15**		**0F**
	16 bits access	Physical Input Registers	Read Input Register	**04**		**04**
		Internal Registers Or Physical Output Registers	Read Holding Registers	**03**		**03**
			Write Single Register	**06**		**06**
			Write Multiple Registers	**16**		**10**
			Read/Write Multiple Registers	**23**		**17**
			Mask Write Register	**22**		**16**
			Read FIFO queue	**24**		**18**
	File record access		Read File record	**20**		**14**
			Write File record	**21**		**15**
Diagnostics			Read Exception status	**07**		**07**
			Diagnostic	**08**	**00-18,20**	**08**
			Get Com event counter	**11**		**OB**
			Get Com Event Log	**12**		**0C**
			Report Server ID	**17**		**11**
			Read device Identification	**43**	**14**	**2B**
Other			Encapsulated Interface Transport	**43**	**13,14**	**2B**
			CANopen General Reference	**43**	**13**	**2B**

APPENDIX 2 (1/3)

Network: 1

MB/TCP communication to external system, control block
and communication block

#DEAMON1
#DEAMON1

EN

M0.3
M: Bit used for initializing, OB100 (Set) and FB10 (Reset)
"Initialize" — INIT

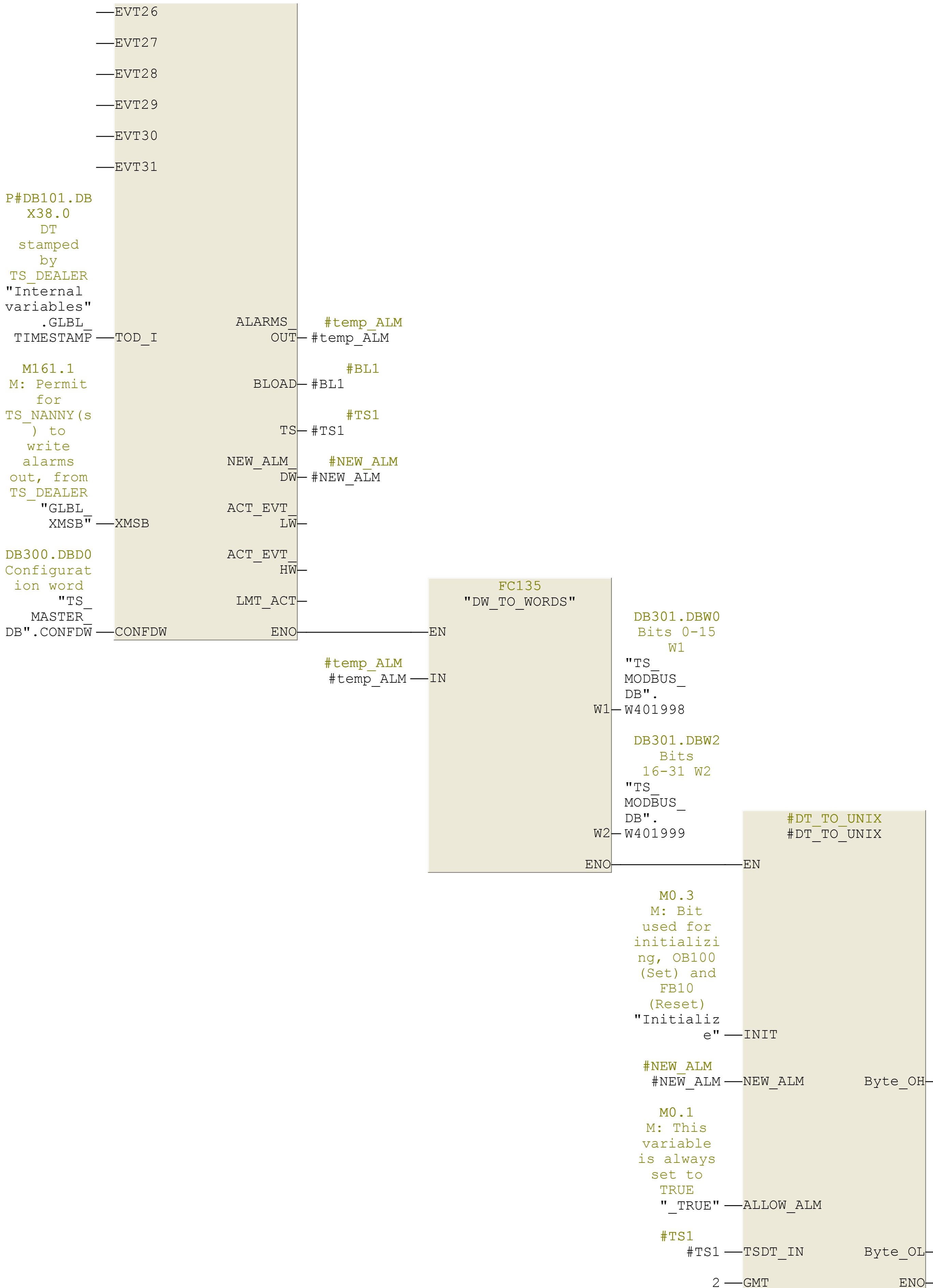
M1.6
M: 1600 ms period
"CLOCKCYCLE_1600" — PULSE

7 — BRST_LMT

EVT0
EVT1
EVT2
EVT3
EVT4
EVT5
EVT6
EVT7
EVT8
EVT9
EVT10
EVT11
EVT12
EVT13
EVT14
EVT15
EVT16
EVT17
EVT18
EVT19
EVT20
EVT21
EVT22
EVT23
EVT24
EVT25

1.A

APPENDIX 2 (2/3)



APPENDIX2 (3/3)

1.C — "TS_MODBUS_DB".W402000

Network: 2

DB355
"Inst_DB_EXT_SYST"
FB182
Supports Modbus functions 3, 4, 6 and 16
"MBTCP_SERVER"

Input actual parameter	Input	Output	Output actual parameter
M0.1 M: This variable is always set to TRUE "_TRUE"	EN	CONN_ON	#TCP_OK #TCP_OK
M0.3 M: Bit used for initializing, OB100 (Set) and FB10 (Reset) "Initialize"	INIT	STAT_CONN	#STAT_C #STAT_C
W#16#6	ID	STAT_R1	#STAT_R1 #STAT_R1
B#16#1	UID	STAT_S1	#STAT_S1 #STAT_S1
T#15S	MONITOR	STAT_FB	#STAT_FB #STAT_FB
W#16#12D	DB_HOLD	ERROR	#Err_e #Err_e
W#16#7CE	DB_HOLD_ADD	RUID	#RUID #RUID
W#16#12D	DB_INPUT	RTI	#RTI #RTI
W#16#7CE	DB_INPUT_ADD	RFUNC	#RFUNC #RFUNC
		START_ADD	#START_ADD #START_ADD
		LENGTH	#LENGTH #LENGTH
		RESBYTES	#RES #RES
		SNDBYTES	#SND #SND
		ENO	

APPENDIX 3

DB301 - <offline> - Declaration view

"TS_MODBUS_DB"
Global data block DB 301

Name: **Family:**
Author: **Version:** 0.1
Block version: 2
Time stamp Code: 03/15/2013 01:27:56 PM
Interface: 03/15/2013 12:45:29 PM
Lengths (block/logic/data): 00320 00196 00000

Block: DB301

Address	Name	Type	Initial value	Comment
0.0		STRUCT		
+0.0	W401998	WORD	W#16#0	Bits 0-15 W1
+2.0	W401999	WORD	W#16#0	Bits 16-31 W2
+4.0	W402000	ARRAY[0..95]	B#16#0	Array containing timestamps for W1
*1.0		BYTE		
+100.0	W402001	ARRAY[0..95]	B#16#0	Array containing timestamps for W2
*1.0		BYTE		
=196.0		END_STRUCT		

APPENDIX 4 (1/3)

```
FUNCTION_BLOCK DT_TO_UNIX

//**********************//
// Declare block inputs //
//**********************//
VAR_INPUT
    INIT                        :BOOL;                  // Initializing
    NEW_ALM                     :BOOL;                  // New alarm occured
    ALLOW_ALM                   :BOOL;                  // Allow alarms to be written to DB
    TSDT_IN                     :ARRAY[0..31] OF DT;    // Array of input DT
    GMT                         :INT;                   // Time zone (+2 in finland)
END_VAR

//***********************//
// Declare block outputs //
//***********************//
VAR_OUTPUT
    Byte_OH                     :ARRAY[0..95] OF BYTE;  //
    Byte_OL                     :ARRAY[0..95] OF BYTE;  //
END_VAR

//**************************//
// Declare static variables //
//**************************//
VAR
    DateAndTime                 :DT;                    // Variable to contain date and time
    DaT AT DateAndTime          :ARRAY[0..7] OF BYTE;   // Array containing hour, minute, second
, etc...

    Temp_Year                   :ARRAY[0..31] OF DINT;  // Year (yy)
    Temp_Month                  :ARRAY[0..31] OF DINT;  // Month (mm)
    Temp_Day                    :ARRAY[0..31] OF DINT;  // Day (dd)
    Temp_Hour                   :ARRAY[0..31] OF DINT;  // Hour
    Temp_Minute                 :ARRAY[0..31] OF DINT;  // Minute
    Temp_Second                 :ARRAY[0..31] OF DINT;  // Second
    Temp_MSecond                :ARRAY[0..31] OF DINT;  // Millisecond
END_VAR

//*************************************//
// Declare temporary (stack) variables //
//*************************************//
VAR_TEMP
    TempH                       :DINT;                  // Temporary holder for unix time
    TempL                       :DINT;                  // Temporary holder for unix time

    LeapYear                    :DINT;                  //

    Year                        :DINT;                  //
    Year_temp                   :DINT;                  //

    A                           :INT;                   //
    B                           :INT;                   //
    C                           :INT;                   //
END_VAR

LABEL
    TS,INITS;
END_LABEL

//*******************//
// Start of the code //
//*******************//
BEGIN

IF INIT THEN
    GOTO INITS;
END_IF;

IF ALLOW_ALM THEN
    GOTO TS;
END_IF;

INITS:  LeapYear := 0;
        Year := 1972;
        TempH := 0;
```

APPENDIX 4 (2/3)

```
7          TempL := 0;
8
9      FOR C := 0 TO 31 DO
0          IF C < 16 THEN
1              Byte_OL[C*6 + 0] := B#16#00;
2              Byte_OL[C*6 + 1] := B#16#00;
3              Byte_OL[C*6 + 2] := B#16#00;
4              Byte_OL[C*6 + 3] := B#16#00;
5              Byte_OL[C*6 + 4] := B#16#00;
6              Byte_OL[C*6 + 5] := B#16#00;
7          ELSIF C >=16 AND C < 32 THEN
8              Byte_OH[(C-16)*6 + 0] := B#16#00;
9              Byte_OH[(C-16)*6 + 1] := B#16#00;
0              Byte_OH[(C-16)*6 + 2] := B#16#00;
1              Byte_OH[(C-16)*6 + 3] := B#16#00;
2              Byte_OH[(C-16)*6 + 4] := B#16#00;
3              Byte_OH[(C-16)*6 + 5] := B#16#00;
4          END_IF;
5      END_FOR;
6  RETURN;
7
8
9
0
1  TS: IF NEW_ALM THEN
2      FOR A := 0 TO 31 DO
3          DateAndTime := TSDT_IN[A];
4
5          Temp_Year[A] := INT_TO_DINT(BCD_TO_INT(BYTE_TO_WORD(DaT[0])));
6          Temp_Month[A] := INT_TO_DINT(BCD_TO_INT(BYTE_TO_WORD(DaT[1])));
7          Temp_Day[A] := INT_TO_DINT(BCD_TO_INT(BYTE_TO_WORD(DaT[2])));
8          Temp_Hour[A] := INT_TO_DINT(BCD_TO_INT(BYTE_TO_WORD(DaT[3])));
9          Temp_Minute[A] := INT_TO_DINT(BCD_TO_INT(BYTE_TO_WORD(DaT[4])));
0          Temp_Second[A] := INT_TO_DINT(BCD_TO_INT(BYTE_TO_WORD(DaT[5])));
1          Temp_MSecond[A] := INT_TO_DINT(BCD_TO_INT(SHL(IN := BYTE_TO_WORD(DaT[6]),N := 4) OR SHR(
   IN := BYTE_TO_WORD(DaT[7]),N := 4)));
2      END_FOR;
3
4
5      FOR B := 0 TO 31 DO
6          Year_temp := Temp_Year[B];
7          Year := 1972;
8          LeapYear := 0;
9
0  //*******************************************//
1  // Calculating amount leap years since 1972 //
2  //*******************************************//
3          WHILE Year < (2000 + Year_temp) DO
4              IF ((((Year MOD 4) = 0) AND NOT (((Year) MOD 100) = 0)) OR (((Year) MOD 400) = 0)) T
   HEN
5                  LeapYear := LeapYear + 1;
6              END_IF;
7              Year := Year + 4;
8          END_WHILE;
9
0  //**********************************************//
1  // Calculating amount of seconds since 1970.1.1 //
2  //**********************************************//
3          IF ((((2000 + Temp_Year[B]) MOD 4) = 0) AND NOT (((2000 + Temp_Year[B]) MOD 100) = 0)) O
   R (((2000 + Temp_Year[B]) MOD 400) = 0)THEN
4              IF Temp_Month[B] > 2 THEN
5                  TempH := (((((2000 + Temp_Year[B]) - 1970) * 365) + LeapYear) + ((367 * Temp_Mon
   th[B] - 362) / 12 + Temp_Day[B] - 2))*86400;
6              ELSE
7                  TempH := (((((2000 + Temp_Year[B]) - 1970) * 365) + LeapYear) + ((367 * Temp_Mon
   th[B] - 362) / 12 + Temp_Day[B]))*86400;
8              END_IF;
9          ELSE
0              IF Temp_Month[B] > 2 THEN
1                  TempH := (((((2000 + Temp_Year[B]) - 1970) * 365) + LeapYear) + ((367 * Temp_Mon
   th[B] - 362) / 12 + Temp_Day[B] - 3))*86400;
2              ELSE
3                  TempH := (((((2000 + Temp_Year[B]) - 1970) * 365) + LeapYear) + ((367 * Temp_Mon
   th[B] - 362) / 12 + Temp_Day[B]))*86400;
4              END_IF;
```

APPENDIX 4 (3/3)

```
5          END_IF;
6
7  //******************************************//
8  // Calculating amount of days since 1970.1.1 //
9  //******************************************//
0          TempL := ((((Temp_Hour[B]) * 3600) - (GMT * 3600)) + ((Temp_Minute[B]-1) * 60) + Temp_Se
   cond[B]) * 1000 + Temp_MSecond[B];
1
2  //************************************************//
3  // Converting the unix time to an array of 6 bytes //
4  //************************************************//
5          IF B < 16 THEN
6              Byte_OL[B*6 + 0] := DWORD_TO_BYTE(SHR(IN := DINT_TO_DWORD(DWORD_TO_DINT(SHR(IN := DI
   NT_TO_DWORD(TempH), N := 16))*1000), N := 24));
7              Byte_OL[B*6 + 1] := DWORD_TO_BYTE(SHR(IN := DINT_TO_DWORD(DWORD_TO_DINT(SHR(IN := DI
   NT_TO_DWORD(TempH), N := 16))*1000), N := 16));
8              Byte_OL[B*6 + 2] := DWORD_TO_BYTE(SHR(IN := DINT_TO_DWORD(TempH*1000 + TempL) ,N :=
   24));
9              Byte_OL[B*6 + 3] := DWORD_TO_BYTE(SHR(IN := DINT_TO_DWORD(TempH*1000 + TempL) ,N :=
   16));
0              Byte_OL[B*6 + 4] := DWORD_TO_BYTE(SHR(IN := DINT_TO_DWORD(TempH*1000 + TempL) ,N :=
   8));
1              Byte_OL[B*6 + 5] := DWORD_TO_BYTE(DINT_TO_DWORD(TempH*1000 + TempL));
2          ELSIF B >=16 AND B < 32 THEN
3              Byte_OH[(B-16)*6 + 0] := DWORD_TO_BYTE(SHR(IN := DINT_TO_DWORD(DWORD_TO_DINT(SHR(IN
   := DINT_TO_DWORD(TempH), N := 16))*1000), N := 24));
4              Byte_OH[(B-16)*6 + 1] := DWORD_TO_BYTE(SHR(IN := DINT_TO_DWORD(DWORD_TO_DINT(SHR(IN
   := DINT_TO_DWORD(TempH), N := 16))*1000), N := 16));
5              Byte_OH[(B-16)*6 + 2] := DWORD_TO_BYTE(SHR(IN := DINT_TO_DWORD(TempH*1000 + TempL) ,
   N := 24));
6              Byte_OH[(B-16)*6 + 3] := DWORD_TO_BYTE(SHR(IN := DINT_TO_DWORD(TempH*1000 + TempL) ,
   N := 16));
7              Byte_OH[(B-16)*6 + 4] := DWORD_TO_BYTE(SHR(IN := DINT_TO_DWORD(TempH*1000 + TempL) ,
   N := 8));
8              Byte_OH[(B-16)*6 + 5] := DWORD_TO_BYTE(DINT_TO_DWORD(TempH*1000 + TempL));
9          END_IF;
0      END_FOR;
1  ELSE
2      RETURN;
3  END_IF;
4  END_FUNCTION_BLOCK
```