Understanding the Modbus Protocol

@version@ (@date@)

by Dieter Wimberger

1. About

This document introduces the reader to the Modbus protocol. It presents a basic protocol description and discusses the serial and the TCP based implementations.

2. Modbus Protocol Basics

Basically Modbus is an application layer protocol (see Figure 1) for communication between devices, mainly to exchange data typical for the field of automation.

Modbus in the ISO/OSI Schema

Table 1: Figure 1: ISO/OSI Context

At this level Modbus is a stateless client-server protocol (e.g. much like HTTP), based on *transactions*, which consist of a *request* (issued by the client) and a *response* (issued by the server). In the field where this protocol is usually applied, there exists a concept that is one of the possible schemas governing the lower level communication behavior on a network using a shared signal cable: Master-Slave. To prevent confusion, the following directed relations describe Master-Slave in terms of the Client-Server paradigm:

- the Master **is a** Client
- the Slave **is a** Server

A transaction and it's context is visualized in Figure 2.

Modbus Transaction

Table 2: Figure 2: Modbus Transaction

The stateless communication is based on a simple package, that is called Protocol Data Unit (PDU). The protocol specification defines three types of PDU's:

- **Request PDU**, consisting of:
  1. a code specifying a function (*Function Code*, 1 byte)
  2. and function specific data (*Function Data*, varying number of bytes)

- **Response PDU**, consisting of:
  1. the function code corresponding to the request (*Function Code*, 1 byte)
  2. and response specific data (*Response Data*, varying number of bytes)
- **Exception Response PDU**, consisting of:
  1. the function code corresponding to the request + 0x80 (128), (*Error Code*, 1 byte)
  2. and a code specifying the exception (*Exception Code*, 1 byte)

Figure 3 presents a visualization of these packages.

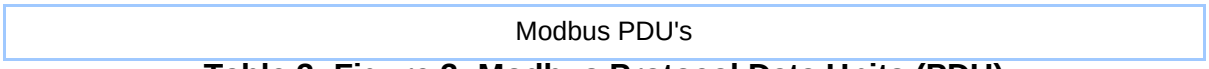

Table 3: Figure 3: Modbus Protocol Data Units (PDU)

2.1. Modbus Functions

The specification defines a certain number of functions, each of which is assigned a specific function code. These are in the range 1-127 (decimal), as 129(1+128)- 255(127+128) represents the range of error codes. While the first published version of the specification defined different classes of functions (e.g. Class 0, Class 1, Class 2), the newly released specification (from http://www.modbus.org; see Knowledge Base Index) defines categories of function codes:

- **Public**
  Are guaranteed to be unique and specify well defined functions that are publicly documented. These are validated by the community and a conformance test exists.
- **User-Defined**
  Are available for user-defined functions, thus their codes might not be unique. The specification defines the code ranges 65-72 and 100-110 for user-defined functions.
- **Reserved**
  These are currently used by some companies for legacy products and are not available for public use (these are not discussed any further in the specification).

The documentation for a function consists of:

1. a description of the function (i.e. what it is good for), it's parameters and return values (including possible exceptions).
2. the assigned *Function Code*
3. the *Request PDU*
4. the *Response PDU*
5. the *Exception Response PDU*

The specification further documents defined and assigned **public** functions.

2.2. Exceptions

In certain cases, the response from a slave will be an exception. The primary identification of an exception response is the error code (function code + 128), which is further specified by the exception code. Assigned codes and descriptions can be found in the specification.

2.3. Modbus Data Model

The basic public functions have been developed for exchanging data typical for the field of automation. Table 1 contains the basic Modbus data types defined by the specification.

Name	Type	Access	Visual
Discrete Input	single bit	read-only	Discrete Input
Discrete Output (Coils)	single bit	read-write	Discrete output/Coil
Input Registers	16-bit word	read-only	Input Register
Holding Registers (Registers)	16-bit word	read-write	(Holding) Register

Table 1: Table 1: Modbus Data Types

Note:

The specification does not define the ways of organizing the related data in a device. However, the organization has a direct influence on the addresses used in basic access functions. (Thus always consult the device's documentation to learn about addressing in basic access functions!)

3. Modbus Implementations

Basically Modbus has been implemented and used over all types of physical links (wire, fiber and radio) and various types of lower level communication stacks. However, we will concentrate on the two basic types of implementations (which are supported by jamod):

1. **Serial**: Asynchronous Master/Slave
2. **IP**: Master/Slave

3.1. Serial Modbus Implementations

Modbus started it's life in form of an implementation for asynchronous serial network communication. The application level protocol operates directly on top of a serial interface and serial communication standards. The most common ones (over wire) are:

- **RS232 (EIA232)**:
  see *The RS232 Standard*
- **RS422/RS485**:
  see *Introduction to RS422 and RS485*

RS232 is used for short distance point-to-point communication, the same is valid for RS 422, which is a bidirectional extension of RS232 for industrial environments, that also supports longer distances.
RS485 can be used for multipoint communication (i.e. multiple devices connected to the same signal cable), employing the Master-Slave paradigm (one master and n fixed address slaves). Figure 4 visualizes the possible network setups.

Serial Network Architectures

Table 1: Figure 4: Serial Network Architectures

To enable the actual communication for this setups, the implementation extends the PDU with additional fields, better said, it wraps the PDU into a package with a *header* and an *error checksum* (see Figure 5). The resulting package is defined by the protocol specification as Application Data Unit (ADU), **that has a maximum package size of 256 bytes**.

Note:

The maximum package size limitation of 256 bytes applies for all existing Modbus protocol implementations!

Modbus Serial ADU

Table 2: Figure 5: Serial ADU

The header is composed of an address field (1 byte) and the tail is an error checksum over the whole package, including the address field (i.e. header). For transmission the Modbus message (i.e. ADU) is placed into a frame that has a known beginning and ending point, allowing detection of the start and the end of a message and thus partial messages. There exist two transmission modes, which differ in encoding, framing and checksum:

1. **ASCII**
   Frames are encoded into two ASCII characters per byte, representing the hexadecimal notation of the byte (i.e. characters 0–9, A–F). The error checksum is represented by a longitudinal redundancy check (LRC; 1 byte) and messages start with a colon (':', 0x3A), and end with a carriage return – line feed ("CRLF", 0x0D0A). Pauses of 1 second between characters can occur.
2. **RTU**
   Frames are transmitted binary to achieve a higher density. The error checksum is represented by a cyclic redundancy check (16 bit CRC; 2 byte) and messages start and end with a silent interval of at least 3.5 character times. This is most easily implemented as a multiple of character times at the baud rate that is being used on the network. The maximum pause that may occur between two bytes is 1.5 character times.

jamod is designed to support both transmission modes, using an implementation which is based on the javax.comm API.

Note:

The RTU implementation does only support the Master side. It is working by the **best effort** principle, which means it might not work in a reliable way in a low-lantency real-time context.

It is indeed possible to implement the serial transport based on other serial stack implementations (i.e. replacements for the Java Comm API implementation) like for example *SerialPort* (http://www.sc-systems.com/products/serialport/serialport.htm). According to the product info it supports around 20 platforms and it has been successfully used to implement the two serial transmission modes in Java (Master only, see Field Talk/Java, a commercial Master protocol pack from Focus Engineering).

3.2. IP based Modbus Implementations

A TCP/IP based Modbus protocol implementation (Modbus/TCP) has been recently committed as an RFC draft to the IETF. It uses the TCP/IP stack for communication (registered port is 502) and extends the PDU with an IP specific header (see Figure 6).
The possible network setups are not governed by the specification; it is possible to setup multi-master systems or realize bidirectional communication (i.e. have nodes that are master and slave at the same time). However, the user should be well aware that there are implications from deviations of the Master/Slave schema.

Modbus TCP ADU

Table 1: Figure 6: Modbus/TCP ADU

The IP specific header (called MBAP in the specification) is 7 bytes long and composed of the following fields:

1. the *invocation identification* (2 bytes) used for transaction pairing; formerly called *transaction identifier*
2. the *protocol identifier* (2 bytes), is 0 for Modbus by default; reserved for future extensions
3. the *length* (2 bytes), a byte count of all following bytes
4. the *unit identifier* (1 byte) used to identify a remote unit located on a non-TCP/IP network

4. Critical Evaluation of the Specification(s)

There are a few points regarding the specification, which are definitely discussable:

1. The specification does not present a consistent naming for all of the basic simple data types. This propagates to the naming of a number basic data access functions. Probably it would be good to elaborate one consistent naming schema, to avoid confusion and allow

better mind mapping.

2. The *Modbus Encapsulated Interface* (MEI) is exposed through a documented public function, without being further explained.
3. The properties of the protocol are perfectly suited for the use of UDP/IP as transport layer protocol:
   1. it is stateless,
   2. transaction oriented
   3. and the package size is limited to 256 bytes, which should be easily transferable over any IP capable link (including IP over Serial) without the necessity to split the package.

   Especially if a real-time communication has to be achieved, it might be of interest to investigate in a Modbus/UDP implementation.

Note:
For learning more about Modbus/UDP, please see: Modbus/UDP Specification.