UDP Slave HOW-TO

@version@ (@date@)

by Dieter Wimberger

1. About

This document is a tutorial for writing Modbus/UDP Slave applications utilizing the *jamod* library. It explains the basics and walk's you through a simple command line Slave implementation,that will serve the values from a static process image on Master requests.
If you are new to Modbus, it is highly recommended to first take a look at *Understanding the Protocol* as well as the actual protocol specification.

Note:
The application build in the tutorial is actually part of the distribution codebase (`net.wimpi.modbus.cmd.UDPSlaveTest`).

2. What is a Slave?

Thinking in terms of the Client-Server network computing paradigm, the Slave application is a Server. It has a *Listener* for receiving an incoming *Request* from the Master application (which indeed is a Client) and sends a corresponding *Response*, just as described in *Understanding the Protocol*.

The simple network setup for this tutorial is composed of two nodes, as depicted in Figure 1.

Network setup

Table 1: Figure 1: Network Setup

The implementation from the *jamod* library will automagically construct the actual responses for requests related to the standard Modbus data model, according to the contents of the actually set *Process Image*.
The reference to the actual *Process Image* is stored in the *Modbus Coupler* a singleton instance accessible throughout the VM.

3. What is a Process Image?

A process image is basically a collection of *Discrete Inputs*, *Discrete Outputs (Coils)*, *Input Registers* and *Registers*.
Please refer to *Understanding the Process Image* for more information.

4. Classes of Interest for the Developer

The motivation for creating *jamod* was to achieve an intuitive and object oriented implementation of the Modbus protocol, in a way, that there is a natural mapping from the domain knowledge (i.e. Modbus protocol) to the abstract class model.
The important elements in the description above (`What is a Slave?`) have been highlighted and the following list represents the mapping between them and the classes from *jamod* that will be needed for a slave implementation:

- *Listener*: `ModbusUDPListener`
- *Process Image*: `ProcessImage` (respectively it's direct known subclass `SimpleProcessImage`)
- *Discrete Inputs*: `DigitalIn` (respectively it's direct known subclass `SimpleDigitalIn`)
- *Discrete Outputs*: `DigitalOut` (respectively it's direct known subclass `SimpleDigitalOut`)
- *Input Registers*: `InputRegister` (respectively it's direct known subclass `SimpleInputRegister`)
- *Registers*: `Register` (respectively it's direct known subclass `SimpleRegister`)
- *Modbus Coupler*: `ModbusCoupler`

5. Implementation

As the idea is to provide a tutorial in form of a very simple command line example, it will consist of only one class and most of the work will be done in the entry method (`public static void main(String args[])`). This is probably not the way *jamod* will be usually employed in OO designs, but we hope it serves the demonstrative purpose.

Now let's start writing code. We need a simple Java application skeleton, with imports of all *jamod* packages:

```
import net.wimpi.modbus.net.*;
import net.wimpi.modbus.procimg.*;
import net.wimpi.modbus.ModbusCoupler;

public class UDPSlaveTest {
  public static void main(String[] args) {
    try {
      ...
      ...
```

```
    } catch (Exception ex) {
      ex.printStackTrace();
    }
  }//main

}//class UDPSlaveTest
```

Next we add the instances and variables the application will need, acquiring the value of the port number from the first commandline parameter if given:

```
/* The important instances and variables */
ModbusUDPListener listener = null;
SimpleProcessImage spi = null;
int port = Modbus.DEFAULT_PORT;

  //1. Set port number from commandline parameter
  if(args != null && args.length ==1) {
    port = Integer.parseInt(args[0]);
  }
```

Next we will construct the process image and setup the coupler to hold the reference:

```
//2. Prepare a process image
spi = new SimpleProcessImage();
spi.addDigitalOut(new SimpleDigitalOut(true));
spi.addDigitalOut(new SimpleDigitalOut(false));
spi.addDigitalIn(new SimpleDigitalIn(false));
spi.addDigitalIn(new SimpleDigitalIn(true));
spi.addDigitalIn(new SimpleDigitalIn(false));
spi.addDigitalIn(new SimpleDigitalIn(true));
spi.addRegister(new SimpleRegister(251));
spi.addInputRegister(new SimpleInputRegister(45));

 //3. Create the coupler holding the image
 ModbusCoupler.createModbusCoupler(spi);
```

Note:

It should be relatively easy to create your own classes of process image related instances. These might even use the Java Native Interface (JNI) to directly access specific hardware, and expose their state as register, input register, input discrete or coil.

The last step is to create a listener with a thread pool size of 3 and start it:

```
//4. Create a listener with 3 threads in pool
listener = new ModbusUDPListener();
listener.setPort(port);
listener.start();
```

That's all, your slave is ready to serve requests.

Note:
The standard port `502` might need special access rights on some operating systems. For tests you might prefer to use some port `>1000`.

You can test the slave we just created using the master application from the *UDP Master How-To*.
The following is an example output from the slave, given the request from the formerly mentioned *UDP Master How-To*.

```
Fangorn:~/development/java/jamod wimpi$ java -Dnet.wimpi.modbus.debug=true \
  -cp build/classes net.wimpi.modbus.cmd.UDPSlaveTest 5555
jModbus Modbus/UDP Slave v0.1
UDPSlaveTerminal::activate()
UDPSlaveTerminal::haveSocket():java.net.DatagramSocket@86db54
UDPSlaveTerminal::addr=:localhost/127.0.0.1:port=5555
UDPSlaveTerminal::receiver started()
UDPSlaveTerminal::sender started()
UDPSlaveTerminal::transport created
UDPSlaveTerminal::activated
Received package to queue.
Request:00 00 00 00 00 06 00 02 00 00 00 04
Response:00 00 00 00 00 04 00 02 01 50
Sent package from queue.
Received package to queue.
Request:00 01 00 00 00 06 00 02 00 00 00 04
Response:00 01 00 00 00 04 00 02 01 50
Sent package from queue.
Received package to queue.
Request:00 02 00 00 00 06 00 02 00 00 00 04
Response:00 02 00 00 00 04 00 02 01 50
Sent package from queue.
```

Note:
The debug outputs of the library can be activated by passing the property `net.wimpi.modbus.debug` and allow to see the actually exchanged modbus messages encoded as hex.