UPDU Modbus TCP

Riedo Networks Ltd
Switzerland

Revision: 3.2.0-e577406e
Date: March 26, 2024

Contents

1 Overview

The UPDU features a *Modbus TCP* server which must be enabled before usage. The *Modbus TCP* protocol allows accessing all measurement values of the device using a very simple and efficient protocol.

The current implementation of the *Modbus TCP* protocol allows reading all relevant values of a device. Writing settings or values is currently not supported.

1.1 Protocol compliancy

The implementation follows the specifications of the following protocol specifications published by the Modbus Organization:

- MODBUS Application Protocol Specification V1.1b3
- MODBUS Messaging on TCP/IP Implementation Guide V1.0b
- MODBUS/TCP Security Protocol Specification V36

Exceptions and conventions to the above mentioned specifications are described below.

1.2 Data types and conventions

In order to overcome the 16 bit upper size limit for values defined by the *Modbus* specification, a few additional datatypes have been implemented.

- `bitfield` : A single Modbus register representing 16 individual bits.
- `uint16` : A single Modbus register holding a 16-bit unsigned integer.
- `uint32` : 2 Modbus registers to hold a 32-bit unsigned integer.
- `uint64` : 4 Modbus registers to hold a 64-bit unsigned integer.
- `int32` : 2 Modbus registers to hold a 32-bit signed integer.

For any integers spanning over multiple Modbus registers, the data is returned in big-endian format. Thus, the lowest register address contains the most significant bits.

Important: The Modbus server supports reading multiple registers with a single read command. Care must be taken when reading values which span over multiple Modbus registers. Data consistency is only guaranteed for a response to a single read command. It is therefore important to ensure that multi-register values are always obtained with a single read instruction.

1.3 Reserved and invalid registers

In addition to the specified and valid registers, certain addresses contain reserved registers. Reserved registers can be read but will not contain valid information. These are are listed as followed:

- `res` : A single register reserved for future or internal use.
- `res[n]` : Multiple (n) registers reserved for future or internal use.

In addition to reserved registers, the global register map defines invalid ranges which are not to be accessed. Any operation on these register addresses will result in returning an error according to the Modbus protocol specification.

2 Supported commands

The following sections describe the currently supported commands. Note that future firmware versions may support additional commands. For details on how to use the commands, refer to the Modbus standards or the documentation of the controller (e.g. PLC).

2.1 Read Device Identification (0x2B/0x0E)

Read device identification (0x0E) is one of the Encapsulated Interface Transport (0x2B) commands. It allows to read identification data from the UPDU.

2.2 Read Input Registers (0x04)

This command is used to read from 1 to 125 continous input registers from an UPDU.

3 Device configuration

In order to use the Modbus TCP server with a UPDU, the device has to be configured accordingly. The following is a quick start using commands which are documented in the *CLI Reference Manual*.

3.1 Unsecure Modbus/TCP

The simplest configuration variant of the Modbus/TCP server on the UPDU is not recommended for use in real world applications. It has neither authentication nor encryption.

It is configured as follows:

```
updu> configure
updu(config)# modbus/tcp
updu(config-modbus/tcp)# enabled
updu(config-modbus/tcp)# transport plain
updu(config-modbus/tcp)# default-role set guest
updu(config-modbus/tcp)# end
Leaving configuration mode. Use the "write" command to save changes.
updu>
```

This causes the Modbus/TCP server to listen on TCP port 502 on all network interfaces. All connections are accepted and are subject to the permissions defined in the `guest` role. Note that the `guest` role is part of the factory configuration, if it has been removed or modified, adjust the `default-role` setting according to the actual configuration.

3.2 Secure Modbus/TCP

The Modbus/TCP server on the UPDU implements "Modbus/TCP Security" which uses X.509 certificates to provide authentication and authorization.

Secure Modbus/TCP works as follows:

1. Upon connecting to the Modbus/TCP server, the server sends its X.509 certificate allowing the client to authenticate the server.
2. The client sends its X.509 certificate allowing the server to authenticate the client. At this point the connection is mutually authenticated.
3. The client certificate can optionally contain the role of the client, which is then used for authorization.

When step 2 is omitted, the connection is not secure. It is encrypted but has no authentication or authorization of the client. This setup is not recommended for use in real world applications.

When only step 3 is omitted, the connection is secure since it is mutually authenticated. All authenticated connections will use the role defined with the `default-role` setting.

3.2.1 Secure Modbus/TCP Certificates

In order to setup secured Modbus/TCP, the following files are required:

- `ca.key` and `ca.crt`: CA key and certificate used for signing the certificates
- `server.key` and `server.crt`: Private key and certificate signed by the CA
- `client.key` and `client.crt`: Private key and certificate signed by the CA

It is possible to have separate CA for signing client and server certificates. For the sake of brevity we are using the same CA to sign both types of certificates here.

The following code snippets illustrate how these certificates can be generated step by step using OpenSSL:

1. Create a CA:

```
openssl genrsa -out ca.key 4096
openssl req -new -x509 -sha256 -key ca.key -out ca.crt -subj '/CN=ModbusTCP Root CA/'
```

2. Create a server certificate and sign it with the CA:

```
openssl genrsa -out server.key 4096
openssl req -new -sha256 -key server.key -out server.csr -subj "/CN=ModbusTCP Server/" -addext "<SAN>"
openssl x509 -req -in server.csr -CA ca.crt -CAkey ca.key -CAcreateserial \
    -out server.crt -days 1000 -sha256 -copy_extensions copy
```

`<SAN>` is for "subject alternative name" and contains the name of the UPDU to which the Modbus/TCP client will connect. It can be a DNS name such as `subjectAltName=DNS:*.mydc.com` but also an IP address like `subjectAltName=IP:10.11.12.13`.

3. Create a client certificate and sign it with the CA:

```
openssl genrsa -out client.key 4096
openssl req -new -sha256 -key client.key -out client.csr -subj '/CN=ModbusTCP Client/' \
    -addext "1.3.6.1.4.1.50316.802.1=ASN1:UTF8String:my-role"
openssl x509 -req -in client.csr -CA ca.crt -CAkey ca.key -CAcreateserial \
    -out client.crt -days 1000 -sha256 -copy_extensions copy
```

This creates a certificate containing the role `my-role` used for authorization. When the `-addext` parameter is omitted, the role configured with the `default-role` command will be used instead.

3.2.2 Configure the UPDU

1. Configure `store-1` with the server key and certificate:

```
updu> configure
updu(config)# certificates
updu(config-certificates)# store-1
updu(config-certificates-store-1)# key-data
Paste private key in PEM format, end with an empty line or CTRL-c or CTRL-d.
```

Insert the contents of the `server.key` file.

```
updu(config-certificates-store-1)# crt-data
Paste certificate in PEM format, end with an empty line or CTRL-c or CTRL-d.
```

Insert the contents of the `server.crt` file.

```
updu(config-certificates-store-1)#
updu(config-certificates-store-1)# exit
updu(config-certificates)#
```

2. Configure `store-2` with the CA certificate:

```
updu(config-certificates)# store-2
updu(config-certificates-store-2)# crt-data
Paste certificate in PEM format, end with an empty line or CTRL-c or CTRL-d.
```

Insert the contents of the `ca.crt` file.

```
updu(config-certificates-store-2)#
updu(config-certificates-store-2)# exit
updu(config-certificates)# exit
updu(config)#
```

3. Configure the Modbus/TCP server:

```
updu(config)# modbus/tcp
updu(config-modbus/tcp)# enabled
updu(config-modbus/tcp)# transport tls
updu(config-modbus/tcp)# certificate store-1
updu(config-modbus/tcp)# auth-certificate store-2
updu(config-modbus/tcp)# default-role none
updu(config-modbus/tcp)# end
Leaving configuration mode. Use the "write" command to save changes.
updu>
```

At this point a client can connect to the Modbus/TCP server securely using the `client.key`, `client.crt` and `ca.crt` files.

4 Print Register Map

As the actual register map depends on the UPDU model, the CLI allows to print a list with the start address of the object registers:

```
updu> show modbus/tcp
Hex     Dec     # Object          Description
0x0000     0  8                   PDU Information
0x0200   512 16 Sensor1           Sensors
0x0210   528 16 Sensor2           Sensors
0x0220   544 16 Sensor3           Sensors
0x0400  1024 32 PDU               PDU Total Power
0x0500  1280 32 Inlet             Inlet Power
0x0600  1536 32 WireL             Wire Power
0x1000  4096 32 Module0           Module Power
0x2000  8192 32 Outlet0.1         Outlet Power
0x2020  8224 32 Outlet0.2         Outlet Power
0x2040  8256 32 Outlet0.3         Outlet Power
0x2060  8288 32 Outlet0.4         Outlet Power
0x2080  8320 32 Outlet0.5         Outlet Power
0x20a0  8352 32 Outlet0.6         Outlet Power
0x20c0  8384 32 Outlet0.7         Outlet Power
0x20e0  8416 32 Outlet0.8         Outlet Power

Legend:
  Hex/Dec: Register address in hexadecimal/decimal notation
  #: Number of registers
  Object: Corresponding PDU object
  Description: Field or register object description
```

To print a complete register map with all registers, append the `detail` parameter:

```
updu-101718> show modbus/tcp detail
Hex     Dec     # Object          Description
0x0000     0  1                   Modbus/TCP register map version
0x0001     1  1                   Number of inlet objects
...
```

5 Device identification

The UPDU supports Basic, Regular and Specific device identification types with Read Device Identification command. Basic (0x01) and Regular (0x02) gives stream access to a group of identification objects whereas Specific (0x04) gives individual object access.

5.1 Device Identification Objects

Device identification objects supported by the UPDU:

Obj ID	Description	category
0x00	Vendor Name	Basic
0x01	Product Code: Model/Serial Number	Basic
0x02	Firmware Revision	Basic
0x03	Vendor Url	Regular
0x04	Product Name	Regular
0x05	Model Name	Regular

6 Holding Register Map (Version 1.0)

The holding registers are read-only registers which can be obtained by using the Read Input Registers command.

6.1 Global Object Mapping

The following object map specifies the address ranges to read values from a UPDU. Depending on the model, a certain number of object are present.

Start	End	Description
0x0000	0x0007	PDU Information object
0x0008	0x01ff	(Reserved range)
0x0200	0x02ff	Sensor objects (16)
0x0300	0x03ff	RCM objects (16)
0x0400	0x041f	PDU Total Power object
0x0420	0x04ff	(Reserved range)
0x0500	0x05ff	Inlet Power objects (8)
0x0600	0x07ff	Wire Power objects (16)
0x0800	0x0fff	Branch Power objects (64)
0x1000	0x17ff	Module Power objects (64)
0x1800	0x1fff	(Reserved range)
0x2000	0x3fff	Outlet Power objects (256)
0x4000	0xffff	(Invalid range)

6.2 PDU Information Object

The *PDU Information Object* lists the active version of the register map along with the number of available objects per type.

Object total size: 8 Registers

Offset	Type	Description
0x00	uint16	Register Map Version
0x01	uint16	Number of inlets
0x02	uint16	Number of wires
0x03	uint16	Number of branches
0x04	uint16	Number of modules
0x05	uint16	Number of outlets
0x06	uint16	Number of RCMs
0x07	uint16	Number of sensors

6.3 Sensor Object

The values of external sensors connected to the device.

Object total size: 16 Registers

Register map:

Offset	Type	Description
0x00	bitfield	Capabilities
0x01	bitfield	Status
0x02	bitfield	Monitoring

Offset	Type	Description
0x03	res	
0x04	uint16	Temperature in 0.1 deg-C
0x05	uint16	Relative humidity 0.1 %RH
0x06	int32	Differential Pressure mPa
0x08	res[10]	

6.3.1 Bitfields

The capabilities, status and monitoring bitfields contain one bit per functionality in this object:

- **Capabilities**: A set bit (1) indicates that a certain functionality supported.
- **Status**: A set bit (1) indicates that the corresponding value is valid. A cleared (0) bit thus indicates an issue with obtaining the measurement.
- **Monitoring**: A set bit (1) indicates that a monitoring condition for the corresponding value is currently active. A clear bit (0) thus indicates that the measurement values are withing the good range or no rules have been configured for the value.

The bits are as follows:

- Bit 0: Sensor provides temperature
- Bit 1: Sensor provides relative humidity
- Bit 2: Sensor provides differential pressure

6.4 RCM Object

Offset	Type	Description
0x00	bitfield	Capabilities
0x01	bitfield	Status
0x02	bitfield	Monitoring
0x03	res	
0x04	uint16	Residual RMS current in 0.1 mA
0x05	uint16	Residual DC current in 0.1 mA
0x06	res[10]	

The capabilities, status and monitoring bitfields contain one bit per functionality in this object:

- The bit in the capabilities bitset tells if the functionality is supported (1) or not (0).
- The bit in the status bitset tells if the corresponding field is currently filled with valid data (1) or not (0).
- The bit in the monitoring bitset tells if the corresponding field is currently OK (0) or if a condition is active (1).

The bits are as follows:

- Bit 0: RCM provides residual RMS current
- Bit 1: RCM provides residual DC current

6.5 Power Object

Offset	Type	Description
0x00	bitfield	Capabilities
0x01	bitfield	Status

Offset	Type	Description
0x02	bitfield	Monitoring
0x03	res	
0x04	uint32	RMS current in mA
0x06	uint32	RMS voltage in mV
0x08	uint32	Active power in W
0x0a	uint32	Apparent power in var
0x0c	uint32	Reactive power in VA
0x0e	uint16	Power factor in per mill
0x0f	uint16	Frequency in 0.001 Hz
0x10	uint64	Positive active energy in Wh
0x14	uint64	Positive reactive energy in varh
0x18	uint64	Negative reactive energy in varh
0x1c	res[4]	

The capabilities, status and monitoring bitfields contain one bit per functionality in this object:

- The bit in the capabilities bitset tells if the functionality is supported (1) or not (0).
- The bit in the status bitset tells if the corresponding field is currently filled with valid data (1) or not (0).
- The bit in the monitoring bitset tells if the corresponding field is currently OK (0) or if a condition is active (1).

The bits are as follows:

- Bit 0: Object provides RMS current
- Bit 1: Object provides RMS voltage
- Bit 2: Object provides active power
- Bit 3: Object provides reactive power
- Bit 4: Object provides apparent power
- Bit 5: Object provides power factor
- Bit 6: Object provides frequency
- Bit 7: Object provides positive active energy
- Bit 8: Object provides positive reactive energy
- Bit 9: Object provides negative reactive energy