

Connectivity Guide

Configuring Universal Device Driver and Profile Library (Modbus Protocol Example)

April 2025
Ref. 1.005

©2021-2025 PTC, Inc. All Rights Reserved.

Table of Contents

1. Introduction 1
2. Key Concepts 1
3. Procedure 2
   - 3.1 Installation 2
   - 3.2 Understanding the Profile 3
   - 3.3 Create a Profile with Kepware Server Configuration 3
   - 3.4 Create a Profile with the Configuration API 4
   - 3.5 Edit the Script 5
   - 3.6 Update the Profile 20
     - 3.6.1 Using Server Config 20
     - 3.6.2 Using the Configuration API 21
   - 3.7 Create a Channel to Use the Profile 21
   - 3.8 Create a Modbus Ethernet Device 22
   - 3.9 Create Tags 23
   - 3.10 Confirm Connectivity to Device 23

 ©2021-2025 PTC, Inc. All Rights Reserved.

1. Introduction

This document aims to assist users in the process of setting up communications between the Universal Device Driver and a device that communicates over the Modbus protocol. This is intended to help provide an understanding of the concepts of the Universal Device driver using Modbus as an example and does not replace the Modbus features in Kepware Server.

This guide assumes an understanding of how the Universal Device driver works and some experience scripting and using JavaScript. For information about the driver, consult the help system (installed with the driver) or available on ptc.com. For help with JavaScript, there are many resources online, such as "Re-introduction to JavaScript", which focuses on the language and not the runtime environment (NodeJs, Chrome/Browser, etc.).

After completing this guide, users should feel comfortable building profiles and communicating with connected devices over custom protocols.

***See Also**: Universal Device Driver User Manual, Profile Library User Manual, Universal Device Driver QuickStart Guide, Universal Device Driver Connectivity Guide (File Access)*

2. Key Concepts

A "profile" exists inside the Profile Library Plug-In and contains a script. Profiles define how the Universal Device driver communicates with a device.

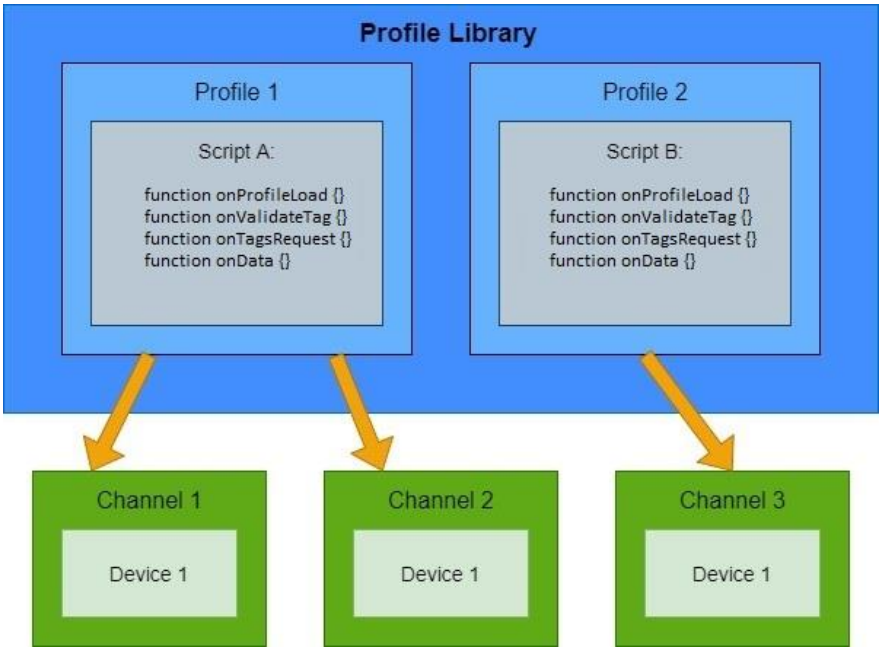
The profile script is a set of JavaScript functions that contain the necessary information to communicate with a device using an arbitrary protocol. The current interface requires four functions to communicate with an Ethernet solicited device: onProfileLoad, onValidateTag, onTagsRequest, and onData. These functions are described in further detail in the respective sections in this document.

See Script Function: onProfileLoad, Script Function: onValidateTag, Script Function: onTagsRequest, and Script Function: onData

Linking is the process of associating a Universal Device channel with a profile to communicate with a device. The image below provides an overview of how a Universal Device project could be structured and what linking a profile to a channel means.

- The Profile Library can contain multiple profiles, each independent of each other.
- Each profile can be linked to multiple channels (1:many).
- Each channel can only be linked to one profile (1:1).
- Each channel maintains its own independent state. Runtime changes (such as creating variables) in Channel1 do *not* affect Channel2, even if they share the same profile script.

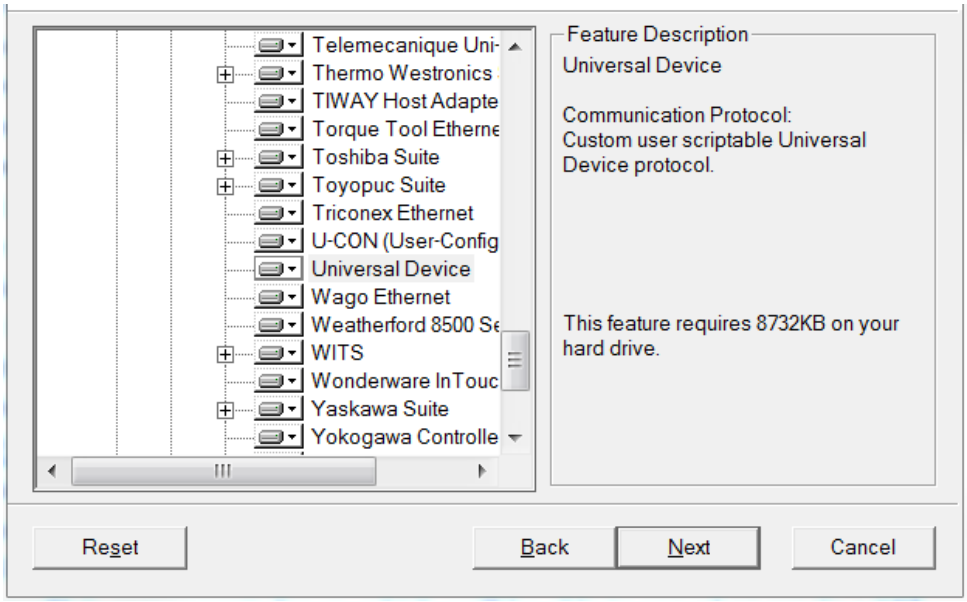
 ©2021-2025 PTC, Inc. All Rights Reserved.



3. Procedure

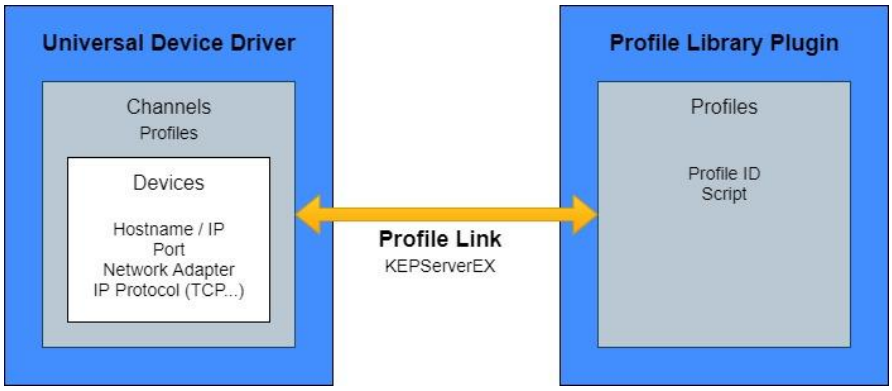
3.1 Installation

The first step is to install Kepware Server with the Universal Device Driver and the Profile Library Plug-In. Selecting the Universal Device option also installs the Profile Library Plug-In.



 ©2021-2025 PTC, Inc. All Rights Reserved.

3.2 Understanding the Profile



Before creating a channel and device and starting communications, a profile must be defined for the channel to link. The profile defines how the channel communicates with devices through a few main components:

Name: The user-facing name of the profile that appears in the Profile Library.

Profile ID: The next important piece of the profile is the Profile ID, formatted as an auto generated GUID. The Profile ID is a unique identifier that the channel uses to establish a link to an existing Profile in the Profile Library. This allows for a project to have many channels linked to the same profile or to have channels linked to different profiles. Once this link is formed, the driver then knows what script it needs to use to communicate with the device.

Script: The most important component of the profile is the JavaScript script property. This script is where to define the protocol-specific communication instructions that the driver will use to communicate with a Modbus device. Define the format and logic to build the data to be sent and received between the server and the target Modbus device.

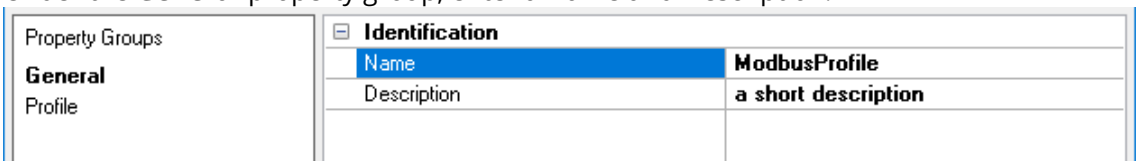
Description: A description of the profile.

3.3 Create a Profile with Kepware Server Configuration

1. Open the server Configuration.
2. Navigate to the Profile Library Plug-in in the left tree.
3. Right-click **Profile Library** and select **Add/New Profile....**



4. Under the **General** property group, enter a Name and Description.



 ©2021-2025 PTC, Inc. All Rights Reserved.

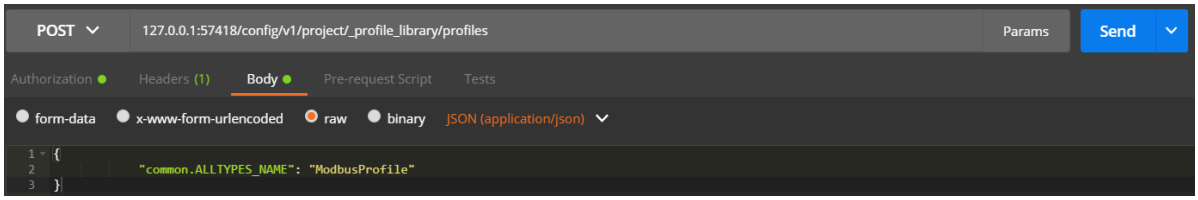
5. Locate the **Profile** property group.
6. Verify the Profile ID and default JavaScript profile script appear automatically.
7. Click **OK**. A profile has been created.

3.4 Create a Profile with the Configuration API

Let's set up an empty profile to start. These steps use Postman (http://getpostman.com). Creating a blank profile fills in the default values of each of the components, which will be visible through a GET on the profile.

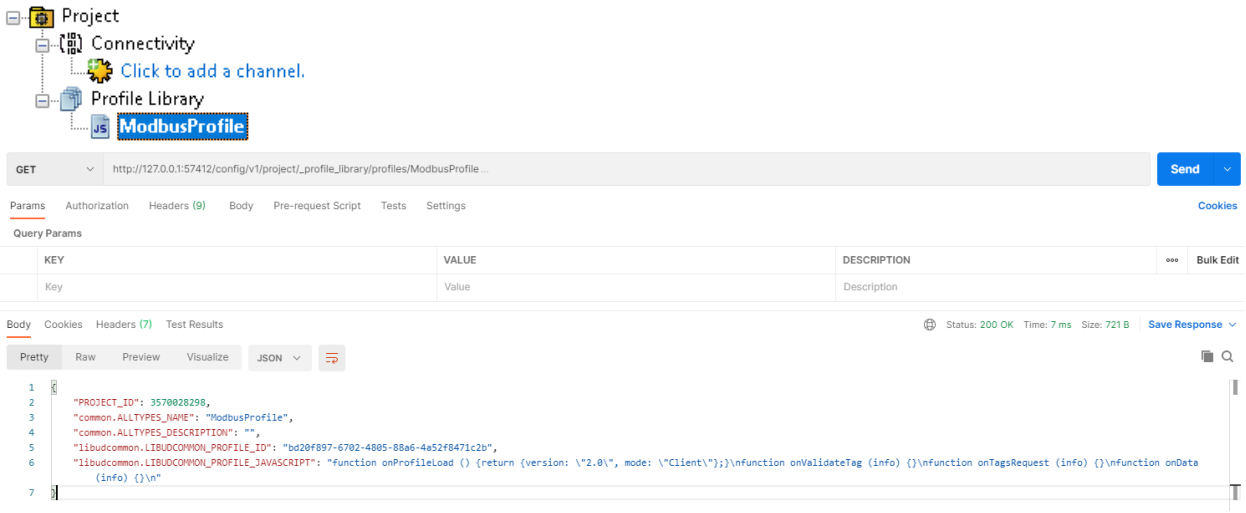
To create a profile access Postman and create a new POST request to the URL:
`<host>:57412/config/v1/project/_profile_library/profiles`
with body:

```
{
    "common.ALLTYPES_NAME": "ModbusProfile"
}
```



Tip: This guide uses ModbusProfile as the profile name.

1. Verify a new entry appears under the Profile Library Plug-In in the Configuration tree or when a GET is executed on the profile by the API endpoint at:
`<host>:57412/config/v1/project/_profile_library/profiles/ModbusProfile`.



2. Notice that the script is filled in with a partial template script with all the JavaScript function definitions necessary to create a profile. The profile ID has been filled in with a new ID.

 ©2021-2025 PTC, Inc. All Rights Reserved.

3.5 Edit the Script

1. Launch a text or code editor to create a new JavaScript file.
2. Review the script one function at a time. There are four functions that must be implemented in the script to support solicited ethernet communications.
   - **onProfileLoad**: Retrieves driver metadata
   - **onValidateTag**: Verifies the address and data type created in the configuration or any dynamic tags created in an OPC client are valid for the end device connected
   - **onTagsRequest**: Builds a packet of bytes to transmit to the device across the wire.
   - **onData**: Interprets the response from the device and updates tag values or indicates if the read or write operation was successful based on the data in the response.

   Notes:

   1. onTagsRequest and onData can do much more than described in this example. These functions can be used to communicate with many kinds of protocols. For more information view the Profile Library Plugin Help documentation.
   2. Another function must be implemented if the Input String device property is populated. See the Profile Library help file for information on the onProfileInputsChange function.
3. Build out the script one function at a time, use the following information to edit the script.

   Required function: onProfileLoad

   The **onProfileLoad** function is the first of these functions called by the driver. It retrieves driver metadata, identifying the interface between the script and the driver by specifying the version of Universal Device Driver with which it was created as well as the mode. For more information on the mode please view the Profile Library plug-in help.

   Note: The only supported version is 2.0. Any other value is rejected, leading to failure of all subsequent functions. Any exception thrown out of any of the "framework" functions is caught and results in failure of the current operation. An exception thrown out of:

   - onProfileLoad causes all subsequent operations to fail until corrected
   - onValidateTag causes the tag address to be treated as "invalid"
   - onTagsRequest causes the read or write operation on the current tag to fail
   - onData causes the read or write operation on the current tag to fail

   Below is the entire onProfileLoad function:

```
function onProfileLoad() {
    return { version: "2.0", mode: "Client" };
}
```

   Required function: onValidateTag

   The **onValidateTag** script function is to validate the address syntax of a tag and the data type, which is central to communicating with a device. In the case of a Modbus device, this function ensures that an address is a holding register in the supported range.

 ©2021-2025 PTC, Inc. All Rights Reserved.

If desired, add logic to this function to modify various tag fields, such as providing a valid default data type, modifying an address format to enforce consistency among tag addresses, assigning a bulkId to group specific tags together, or modifying the data type to a supported array data type and assign the number of columns for the array.

For the onValidateTag function in this Modbus example, review the sections:

```
// Validate the address is a holding register in the supported range
let tagAddress = info.tag.address;

// Regex to parse and capture:
// <5-digit numeric address> for non-array tag. For example, 40001
// <5-digit numeric address>[<1-4 digit array columns>] for array tag. For example,
40001[10] or 40001[120]
let addressRegex = /^(\d{5})(?:\[(\d{1,4})\])?$/;

let numericAddress = 0;
let arrayColumns = 0;
const match = tagAddress.match (addressRegex);
if (match) {
    if (match.length == 3){
        // First result is the match, second result is the register address, third result
is the array columns (if captured)
        numericAddress = parseInt (match[1], 10);
        if (isNaN (match[2])){  // Non-array
            log(`Modbus Ethernet onValidateTag: Tag address ${tagAddress} parse succeeded
=> numeric address = ${numericAddress}`, VERBOSE_LOGGING);
        } else {    // Array
            arrayColumns = parseInt (match[2], 10);
            log(`Modbus Ethernet onValidateTag: Tag address ${tagAddress} parse succeeded
=> numeric address = ${numericAddress}, array columns = ${arrayColumns}`,
VERBOSE_LOGGING);
        }
    } else {
        log(`Modbus Ethernet onValidateTag: Unexpected regex match with ${match.length}
results.`, VERBOSE_LOGGING);
        info.tag.valid = false;
        return info.tag;
    }
} else {
    log(`Modbus Ethernet onValidateTag: Tag address ${tagAddress} parse failed`,
VERBOSE_LOGGING);
    info.tag.valid = false;
    return info.tag;
}
```

The code above offers a look at the JavaScript object `info` the driver provides to the script writer. This object is meant to hold data to be exchanged between the script and the driver.

 ©2021-2025 PTC, Inc. All Rights Reserved.

In this example, the script expects the address received from the driver (`info.tag.address`) to indicate whether to assign an array column value in the `info.tag.arrayColumns`. The script verifies the main address is 5 digits. If there is a value within square brackets; this is an array column definition and it must be 1 to 4 digits. If the address is not formatted properly, the script fails the tag being added by setting the valid field of the tag to false: `info.tag.valid = false` and the tag is returned. If the regular expression pattern matches, further validation of the tag address occurs.

```
try {
    if (numericAddress < MINREGISTERADDRESS || numericAddress > MAXREGISTERADDRESS ||
isNaN(numericAddress)) {
        log(`Modbus Ethernet onValidateTag: Tag address ${tagAddress} out of range`,
VERBOSE_LOGGING);
        info.tag.valid = false;
        return info.tag;
    }

    // arrayColumns will be validated by driver.
    // Note: Don't set info.tag.arrayColumns if arrayColumns is 0
    if (arrayColumns > 0) {
        info.tag.arrayColumns = arrayColumns;
    }

    // Provide a valid default data type based on array vs non-array
    // Note:  "Default" is an invalid data type
    if (info.tag.dataType === "Default"){
        if (arrayColumns > 0){
            info.tag.dataType = "Word Array"
        } else {
            info.tag.dataType = "Word"
        }
    }

    // If grouping tags into bulks, assign bulkId now.
    // Otherwise, the next bulkId is assigned by default.
    let bulkId = Math.floor((numericAddress - MINREGISTERADDRESS)/BULKREGISTERCOUNT);
    info.tag.bulkId = bulkId;
    log(`Modbus Ethernet onValidateTag: Bulk register count ${BULKREGISTERCOUNT}, address
${tagAddress}, bulkId ${info.tag.bulkId}`, VERBOSE_LOGGING);

    info.tag.valid = true;
    return info.tag;
}
catch (e) {
    // Use log to provide helpful information that can assist with error resolution
    log(`Unexpected error (onValidateTag): ${e.message}`, STD_LOGGING);
    info.tag.valid = false;
    return info.tag;
}
```

 ©2021-2025 PTC, Inc. All Rights Reserved.

The above code checks if the main address is in the expected range for a Modbus holding register as defined by constants `MINREGISTERADDRESS, MAXREGISTERADDRESS`. If it's not in that range, the script fails the tag being added by setting the valid field of the tag to false: `info.tag.valid = false` and returns the tag.

The code also provides examples of logic to modify various tag fields, such as `info.tag.arrayColumns`, `info.tag.dataType`, and `info.tag.bulkId`.

If the tag address contained an array columns value within the square brackets, then the script sets the `info.tag.arrayColumns` to that value. The driver will further validate the array columns value depending on the tag data type.

If Default is initially selected for the tag data type, then this example script sets it to `Word` or `Word Array` depending on whether the tag address provided indicated it is an array or not. While Default is a Kepware server data type, it is an invalid return value for a tag data type (i.e., `info.tag.dataType`). As such, provide an appropriate and valid data type if the data type is set as Default.

This example script also defines the bulkId field for each tag. The register in the address along with the `BULKREGISTERCOUNT` constant facilitates assigning the bulkId that allows blocking together consecutive registers. Once the tags are blocked together, the Universal Device driver will then provide them in the tags object passed to the onTagsRequest and onData functions.

Below is the entire onValidateTag function:

```
function onValidateTag(info) {
    // Validate the address is a holding register in the supported range
    let tagAddress = info.tag.address;

    // Regex to parse and capture:
    // <5-digit numeric address> for non-array tag. For example, 40001
    // <5-digit numeric address>[<1-4 digit array columns>] for array tag. For example,
40001[10] or 40001[120]
    let addressRegex = /^(\d{5})(?:\[(\d{1,4})\])?$/;

    let numericAddress = 0;
    let arrayColumns = 0;
    const match = tagAddress.match (addressRegex);
    if (match) {
        if (match.length == 3){
            // First result is the match, second result is the register address, third
result is the array columns (if captured)
            numericAddress = parseInt (match[1], 10);
            if (isNaN (match[2])){  // Non-array
                log(`Modbus Ethernet onValidateTag: Tag address ${tagAddress} parse
succeeded => numeric address = ${numericAddress}`, VERBOSE_LOGGING);
            } else {    // Array
                arrayColumns = parseInt (match[2], 10);
```

 ©2021-2025 PTC, Inc. All Rights Reserved.

```
                log(`Modbus Ethernet onValidateTag: Tag address ${tagAddress} parse
succeeded => numeric address = ${numericAddress}, array columns = ${arrayColumns}`,
VERBOSE_LOGGING);
            }
        } else {
              log(`Modbus Ethernet onValidateTag: Unexpected regex match with
${match.length} results.`, VERBOSE_LOGGING);
            info.tag.valid = false;
            return info.tag;
        }
    } else {
        log(`Modbus Ethernet onValidateTag: Tag address ${tagAddress} parse failed`,
VERBOSE_LOGGING);
        info.tag.valid = false;
        return info.tag;
    }

    try {
        if (numericAddress < MINREGISTERADDRESS || numericAddress > MAXREGISTERADDRESS ||
isNaN(numericAddress)) {
            log(`Modbus Ethernet onValidateTag: Tag address ${tagAddress} out of range`,
VERBOSE_LOGGING);
            info.tag.valid = false;
            return info.tag;
        }

        // arrayColumns will be validated by driver.
        // Note: Don't set info.tag.arrayColumns if arrayColumns is 0
        if (arrayColumns > 0) {
            info.tag.arrayColumns = arrayColumns;
        }

        // Provide a valid default data type based on array vs non-array
        // Note:  "Default" is an invalid data type
        if (info.tag.dataType === "Default"){
            if (arrayColumns > 0){
                info.tag.dataType = "Word Array"
            } else {
                info.tag.dataType = "Word"
            }
        }

        // If grouping tags into bulks, assign bulkId now.
        // Otherwise, the next bulkId is assigned by default.
        let bulkId = Math.floor((numericAddress - MINREGISTERADDRESS)/BULKREGISTERCOUNT);
        info.tag.bulkId = bulkId;
```

 ©2021-2025 PTC, Inc. All Rights Reserved.

```
        log(`Modbus Ethernet onValidateTag: Bulk register count ${BULKREGISTERCOUNT},
address ${tagAddress}, bulkId ${info.tag.bulkId}`, VERBOSE_LOGGING);

        info.tag.valid = true;
        return info.tag;
    }
     catch (e) {
        // Use log to provide helpful information that can assist with error resolution
        log(`Unexpected error (onValidateTag): ${e.message}`, STD_LOGGING);
        info.tag.valid = false;
        return info.tag;
    }
}
```

Required function: onTagsRequest

The **onTagsRequest** script function builds a packet of bytes that is sent to the target Modbus device. In the example implementation, the onTagsRequest function makes use of two helper functions to build action-specific packet: BuildReadMessage and BuildWriteMessage:

```
function onTagsRequest(info) {
let action = "Fail";

if (info.type === "Read") {
    let readData = BuildReadMessage(info.tags);

    // Evaluate if the data was successfully built
    if (readData.length === 12) {
        action = "Receive";
    }

    return { action: action, data: readData };

} else if (info.type === "Write") {
    SENTWRITEDATA = BuildWriteMessage(info.tags);

    // Evaluate if the data was successfully built
    if (SENTWRITEDATA.length > 0) {
        action = "Receive";
    }

    return { action: action, data: SENTWRITEDATA };
}
}
```

Unlike the onTagsRequest function, these helper functions are not required; they help make the script more manageable. Let's dive into these helper functions now.

 ©2021-2025 PTC, Inc. All Rights Reserved.

Helper Function: BuildReadMessage

This function builds into the packet the function code for a Modbus read to ensure that the read is on the appropriate address(es). Most of the Modbus-specific pieces of this snippet are documented in code comments with the important parts called out.

The Modbus protocol supports blocking / bulk read and write functionality. The Universal Device Driver supports blocking non-array tags for reads but does not support blocking tags for writes. The tags parameter is an array of tag objects containing at least one tag element. If, in onValidateTag, the script assigned the same bulkId to more than one non-array tag, then those tags sharing a bulkId are included in the tags object when the request type is Read.

```
// This should never happen, but it's best practice
if (tags.length === 0) {
    throw "No tags were requested for read request.";
}

let first = 0;  // To contain the lowest register, zero based.
let count = 0;  // To contain the number of registers required to read.

if (tags[0].arrayColumns){
    if (tags.length > 1) {
        throw "Only one array tag expected";
    }
    // Array read
    first = parseInt(tags[0].address, 10) - MINREGISTERADDRESS;
    count = tags[0].arrayColumns;
} else {
    // Block read; sort the Modbus registers low to high
    let registers = [];
    for (let i=0; i < tags.length; i++) {
        registers[i] = parseInt(tags[i].address, 10);
    }
    registers.sort (sortNumber);

    // Find the lowest register, and the number of registers required to read the whole
block
    first = registers[0];
    count = registers[registers.length - 1] - first + 1;
    // Get the zero based register index to make the request
    first -= MINREGISTERADDRESS;
}
```

The code above checks the tags component of the JavaScript object info (i.e. `info.tags`). This component holds an array of tag objects. Each tag has an address used to build a request packet for a read. The beginning of this section of code ensures that the driver has given a tag to build a request packet. If the length of the tags array object is zero, it exits the function because there's no reason for the driver to build a request packet if no tag – and in turn, no address – is provided. Furthermore, if there is more than one tag in the tags object and the arrayColumns

 ©2021-2025 PTC, Inc. All Rights Reserved.

field is defined for the first tag, then the script exits the function because the driver should only provide one array tag in the tags object.

The variables indicating the first register address and the count of register addresses are assigned values depending on whether the tags object contained one array tag or one or more tags that shared a bulkId.

```
// Update the transaction ID in the stateful transaction object
if (TXID === undefined) {
    TXID = 0;
} else {
    TXID++;
}
```

JavaScript is not a strongly typed language, making it possible to modify a variable's type or composition at runtime. This is something to take advantage of within the BuildReadMessage function. The above code snippet updates the value of a global variable TXID, which represents a transaction ID exchanged between the script and the driver. Use this global variable to keep track of the number of times it is building packets to transmit to the device. It's important to keep track of this because the transaction ID is a necessary part of the Modbus protocol, as seen in the next step. TXID is stateful between transactions because it is shared between the script and driver and maintains state across transactions. Every time this function is called, the transaction ID value maintains the state it was the last time it was changed at runtime.

```
// Build the Modbus Ethernet data
let data =
// ----Transaction ID------|-Protocol--|---Length--|Server|-Fxn-|
[hiByte(TXID), loByte(TXID), 0x00, 0x00, 0x00, 0x06, 0x00, 0x03,
------Starting Address-------|-------Register count--------|
hiByte(first), loByte(first), hiByte(count), loByte(count)]
```

The above shows the packet being constructed. It is an array of bytes to be sent to the Modbus device. The code comments the different parts of the packet that are defined in the Modbus protocol; for instance, the TXID described earlier is used in the protocol as the top two bytes.

- **Note**: Only bytes are currently supported for the data array.

Below is the entire BuildReadMessage function:

```
function BuildReadMessage (tags) {
    // This should never happen, but it's best practice
    if (tags.length === 0) {
        throw "No tags were requested for read request.";
    }

    let first = 0;  // To contain the lowest register, zero based.
    let count = 0;  // To contain the number of registers required to read.

    if (tags[0].arrayColumns){
        if (tags.length > 1) {
```

 ©2021-2025 PTC, Inc. All Rights Reserved.

```
            throw "Only one array tag expected";
        }
        // Array read
        first = parseInt(tags[0].address, 10) - MINREGISTERADDRESS;
        count = tags[0].arrayColumns;
    } else {
        // Block read; sort the Modbus registers low to high
        let registers = [];
        for (let i=0; i < tags.length; i++) {
            registers[i] = parseInt(tags[i].address, 10);
        }
        registers.sort (sortNumber);

        // Find the lowest register, and the number of registers required to read the
whole block
        first = registers[0];
        count = registers[registers.length - 1] - first + 1;
        // Get the zero based register index to make the request
        first -= MINREGISTERADDRESS;
    }

    // Update the transaction ID in the stateful transaction object
    if (TXID === undefined) {
        TXID = 0;
    } else {
        TXID++;
    }

    // Build the Modbus Ethernet data
    let data =
    // ----Transaction ID------|-Protocol--|---Length--|Server|-Fxn-|------Starting
Address-------|-------Register count--------|
    [hiByte(TXID), loByte(TXID), 0x00, 0x00, 0x00, 0x06, 0x00, 0x03, hiByte(first),
loByte(first), hiByte(count), loByte(count)]

    log(`Modbus Ethernet BuildReadMessage: data = [${data}]`, VERBOSE_LOGGING);
    return data;
}
```

Helper Function: BuildWriteMessage

The BuildWriteMessage function is similar to the BuildReadMessage function in that it assists with building an array of bytes to send the device. However, this function facilitates writing a value to, rather than reading a value from, a Modbus device.

- **Note**: Not all devices support writes. If the target device does support writes, the BuildWriteMessage function – in conjunction with the ParseWriteMessage function – provides an example of how to implement this functionality.

```
// This should never happen but it's best practice
```

 ©2021-2025 PTC, Inc. All Rights Reserved.

```
if (tags.length === 0) {
    throw "No tags were requested for write request.";
}

// Sort the Modbus registers low to high
let register = parseInt(tags[0].address, 10);
register -= MINREGISTERADDRESS;

// Get the value to write which is located in the first
// element in the tags[n].value object
// Note: For array tags, value is an array.
let value = tags[0].value;
log(`Modbus Ethernet BuildWriteMessage: Value being written for address ${tags[0].address}
= ${value}`, VERBOSE_LOGGING);
```

The code above assigns the integer value of the tag address to the variable register. Additionally, is assigns the value of the first tag value to the variable value since KEPServerEX only allows single writes.

```
// Update the transaction ID in the stateful transaction object
if (TXID === undefined) {
    TXID = 0;
} else {
    TXID++;
}
let data = [];
if (tags[0].arrayColumns){
    let first = register;

    // Build the Modbus Ethernet data (Multiple Register Write)
    data =
        // ----Transaction ID-----|-Protocol--|---Length--|Server|-Fxn-|--------Starting
Address----|-Reg Count-|-Byte Count-|--values to write (added separately)--|
        [
        hiByte(TXID), loByte(TXID), 0x00, 0x00, 0x00, 0x00, 0x00, 0x10, hiByte(first),
loByte(first), 0x00, 0x00, 0x00
        ];

    let length = 0;                 // Running number of function bytes
    let count = value.length;       // Number of registers

    // Loop 2-byte register values in the array and push into data frame
    for (let i=0; i < count; i++) {
        data.push (hiByte(value[i]));
        data.push (loByte(value[i]));
        length += 2;
    }
```

 ©2021-2025 PTC, Inc. All Rights Reserved.

```
    // Set register count in data frame
    data [10] = hiByte(count);
    data [11] = loByte(count);

    // Set function byte count
    data[12] = length;

    // Update length to be function byte count + header bytes
    length += 7;    // Server, Fxn, Starting Address, Register Count, and Byte Count

    // Set function length
    data [4] = hiByte(length);
    data [5] = loByte(length);
    log(`Modbus Ethernet BuildWriteMessage (multiple register write): data = [${data}]`,
VERBOSE_LOGGING);
} else {
    // Build the Modbus Ethernet data (Single Register Write)
    data =
        // ----Transaction ID-----|-Protocol--|---Length--|Server|-Fxn-|--------Starting
Address----------|-------value to write--------|
        [
        hiByte(TXID), loByte(TXID), 0x00, 0x00, 0x00, 0x06, 0x00, 0x06, hiByte(register),
loByte(register), hiByte(value), loByte(value)
        ];

    log(`Modbus Ethernet BuildWriteMessage (single register write): data = [${data}]`,
VERBOSE_LOGGING);
}

return data;
```

The above shows how to build up a write packet, which is very similar to building a read packet within the BuildReadMessage function. The packet data built depends on if the tags object contains one array tag or multiple tags that share the same bulkId. If the first tag defines an arrayColumns, then the data is built with the Modbus multiple register write function code, otherwise it has the Modbus single register write function code.

Required function: onData

The **onData** script function parses the array of bytes received from a Modbus device. In the example implementation, as was the case with the **onTagsRequest** function, the **onData** function uses two helper functions to parse responses from a Modbus device: ParseReadMessage and ParseWriteMessage:

```
function onData(info) {
    let action = ACTIONFAILURE;
```

 ©2021-2025 PTC, Inc. All Rights Reserved.

```
    if (info.type === "Read") {
        let tags = ParseReadMessage(info.tags, info.data);

        // Evaluate if the data was successfully parsed from the packet
        if (tags[0].value != null || tags[0].quality != null) {
            action = ACTIONCOMPLETE;
        }

        return { action: action, tags: tags };

    } else if (info.type === "Write") {
        action = ParseWriteMessage(info.data);
        return { action: action, tags: info.tags };
    }
}
```

Helper Function: ParseReadMessage

This function's purpose is to parse an incoming packet into a tag value to update the respective tag in the server. The incoming packet is passed to the script via the JavaScript object information as the returned byte array is contained in its data component (i.e. `info.data`). The function determines what information is important based on the protocol specification and extracts the value for the tag/address. This value is assigned to the value field of the tag (e.g. `info.tags[0].value`) and then returned from the function, which is how the tag is updated in the server.

```
function ParseReadMessage(tags, data) {
    // This should never happen but it's best practice
    if (tags.length === 0) {
        throw "No tags were requested for read request.";
    }

    log(`Modbus Ethernet ParseReadMessage: data = ${JSON.stringify(data)}`,
VERBOSE_LOGGING);

    let count = 0;
    if (tags[0].arrayColumns){
        count = tags[0].arrayColumns
    } else {
        count = tags.length;
    }

    // MBE Response values start here:
    let offset = 9;
    // Enough bytes?
    if (data.length < offset + 2 * count) {
        // Log message only once for this response
        if (data.length === offset){
```

 ©2021-2025 PTC, Inc. All Rights Reserved.

```
            log(`Modbus Ethernet ParseReadMessage: Device returned an error code
${data[7]}, ${data[8]}`);
         } else {
            log(`Modbus Ethernet ParseReadMessage: Invalid response from device`);
        }

        // Iterate the registers and set the quality of each tag to bad
        for (let i = 0; i < tags.length; ++i) {
            tags[i].quality = "Bad";
        }
    } else {
        if (tags[0].arrayColumns){
            tags[0].value = []; // Initialize as array so values can be pushed on

            // Populate tag array value
            for (let i = 0; i < count; ++i) {
                let hi = data[2*i + offset];
                let lo = data[2*i + offset + 1];
                tags[0].value.push (wordFromBytes (hi, lo));
            }

            log(`Modbus Ethernet ParseReadMessage: Value read for address
${tags[0].address} = [${tags[0].value}]`, VERBOSE_LOGGING);
        } else {
            // Convert the string addresses to integers (for example, 40001)
            let registers = [];
            for (let i=0; i < tags.length; i++) {
                registers[i] = parseInt(tags[i].address, 10);
            }

            // Find the lowest numbered register - this is the starting address
            let startingAddress = Array.min (registers);
```

The code above performs error checking and gathering some information about the transaction. If the number of bytes in the response is not the number of bytes expected, then the script sets the quality of each tag to Bad. If the response appears to include an error code from the device, then the script provides that information in the message passed to the log function. Otherwise, the script logs a message indicating an invalid response from the device. The result is an updated tags component of the JavaScript object info to be shared with the driver and ultimately used to update the tag qualities in the server.

```
// Iterate the registers and lookup the response value for each
// Assigning the quality of the tag is optional. If undefined, Good is assumed.
for (let i = 0; i < registers.length; ++i) {
    // Calculate the index of this register's value in the response buffer
    let index = registers[i] - startingAddress;
    // Extract it from the response buffer
```

 ©2021-2025 PTC, Inc. All Rights Reserved.

```
    let hi = data[2*index + offset];
    let lo = data[2*index + offset + 1];
    tags[i].value = (wordFromBytes (hi, lo));
}
```

The code above extracts the value returned from the device and assigns it to the appropriate tag to be used to update the tag value in the server. The result is an updated tags component of the JavaScript object info to be shared with the driver and ultimately used to update the tag values in the server.

Below is the entire ParseReadMessage function.

```
function ParseReadMessage(tags, data) {
    // This should never happen but it's best practice
    if (tags.length === 0) {
        throw "No tags were requested for read request.";
    }

    log(`Modbus Ethernet ParseReadMessage: data = ${JSON.stringify(data)}`,
VERBOSE_LOGGING);

    let count = 0;
    if (tags[0].arrayColumns){
        count = tags[0].arrayColumns
    } else {
        count = tags.length;
    }

    // MBE Response values start here:
    let offset = 9;
    // Enough bytes?
    if (data.length < offset + 2 * count) {
        // Log message only once for this response
        if (data.length === offset){
            log(`Modbus Ethernet ParseReadMessage: Device returned an error code
${data[7]}, ${data[8]}`);
        } else {
            log(`Modbus Ethernet ParseReadMessage: Invalid response from device`);
        }

        // Iterate the registers and set the quality of each tag to bad
        for (let i = 0; i < tags.length; ++i) {
            tags[i].quality = "Bad";
        }
    } else {
        if (tags[0].arrayColumns){
            tags[0].value = []; // Initialize as array so values can be pushed on
```

 ©2021-2025 PTC, Inc. All Rights Reserved.

```
            // Populate tag array value
            for (let i = 0; i < count; ++i) {
                let hi = data[2*i + offset];
                let lo = data[2*i + offset + 1];
                tags[0].value.push (wordFromBytes (hi, lo));
            }

            log(`Modbus Ethernet ParseReadMessage: Value read for address
${tags[0].address} = [${tags[0].value}]`, VERBOSE_LOGGING);
        } else {
            // Convert the string addresses to integers (for example, 40001)
            let registers = [];
            for (let i=0; i < tags.length; i++) {
                registers[i] = parseInt(tags[i].address, 10);
            }

            // Find the lowest numbered register - this is the starting address
            let startingAddress = Array.min (registers);

            // Iterate the registers and lookup the response value for each.
            // Assigning the quality of the tag is optional. If undefined, Good is
assumed.
            for (let i = 0; i < registers.length; ++i) {
                // Calc the index of this register's value in the response buffer
                let index = registers[i] - startingAddress;

                // Extract the value from the response buffer
                let hi = data[2*index + offset];
                 let lo = data[2*index + offset + 1];
                tags[i].value = (wordFromBytes (hi, lo));
                log(`Modbus Ethernet ParseReadMessage: Value read for address
${tags[i].address} = ${tags[i].value}`, VERBOSE_LOGGING);
            }
```

```
        }
    }

    return tags;
}
```

Helper Function: ParseWriteMessage

The purpose of the ParseWriteMessage function is to determine if the write was successful. Most devices respond that the request was received and executed. In the case of Modbus, the response echoes the request, which makes it possible to compare the returned message with the sent message that was saved in the global variable SENTWRITEDATA.

Below is the entire ParseWriteMessage function:

 ©2021-2025 PTC, Inc. All Rights Reserved.

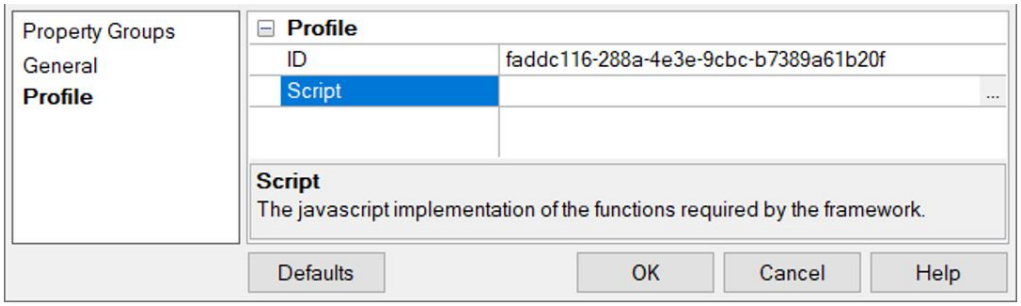
```
function ParseWriteMessage(data) {
// Modbus echoes a write request so if the data sent
// does not match the data received, then the write fails
SENTWRITEDATA.forEach((e1) => data.forEach((e2) => {
    if (e1 !== e2) {
        return "Fail";
    }
}));
return "Complete";
}
```

3.6 Update the Profile

Now that script implementation is complete, replace the template script in the profile with the finalized script.

3.6.1 Using Server Config

1. Open the property editor for the new profile created in this exercise.
2. Load the profile in the **Script** field by copying the content of entire JavaScript file and paste it into the **Script** field in the property editor OR clicking the ellipses (**...**) at the right of the Script text box to launch a file browser to locate the JavaScript file and click **OK**.



> **Note**: The property editor only accepts multi-line text that uses Linux line endings (\n character) and not Windows line endings (\r\n characters). If a pasted script appears with only the first line, use your text editor to remove all line endings or convert them to Linux (\n character) line endings.

3. Click **OK**.

 ©2021-2025 PTC, Inc. All Rights Reserved.

3.6.2 Using the Configuration API

1. Create a PUT to the profile at URL

   ```
   <host>:57412/config/v1/project/_profile_library/profiles
   ```

   with body:

   ```
   {
        "FORCE_UPDATE": true,
        "libudcommon.LIBUDCOMMON_PROFILE_JAVASCRIPT": <ESCAPED SCRIPT>
   }
   ```

2. This script is put into the PUT body where <ESCAPED SCRIPT> is. Note is that this script needs to be 'escaped' before being pasted in; JSON characters (e.g. many symbols, including, but not limited to '\' '/') need to be marked with quotes so that they are interpreted as plain text and not as JSON formatting characters.

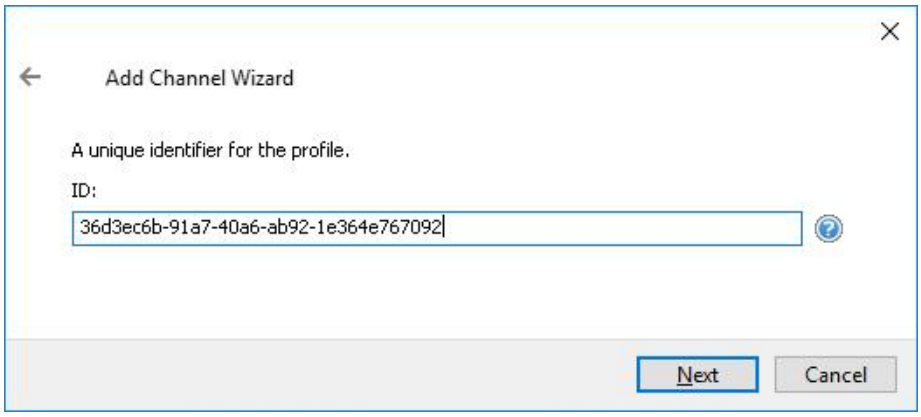
   - **Tip**: There are many tools online that can be used to complete such a task, such as https://www.freeformatter.com/json-escape.html.

3. Execute the PUT request.
4. The profile is created and ready to start communicating with the device.
5. Re-initialize the server.

3.7 Create a Channel to Use the Profile

The next step is to configure a channel using the Universal Device driver. This guide leaves most settings at the defaults.

1. Create a new Universal Device channel (named Universal Device in this guide).
2. Click **Next** through to the Profile ID page of the channel creation wizard.
3. Ensure the new channel is linked to the new profile just created.
4. The Profile ID value is a randomly generated GUID and should be copied from the ID field of the profile (in this example case 36d3ec6b-91a7-40a6-ab92-1e364e767092).



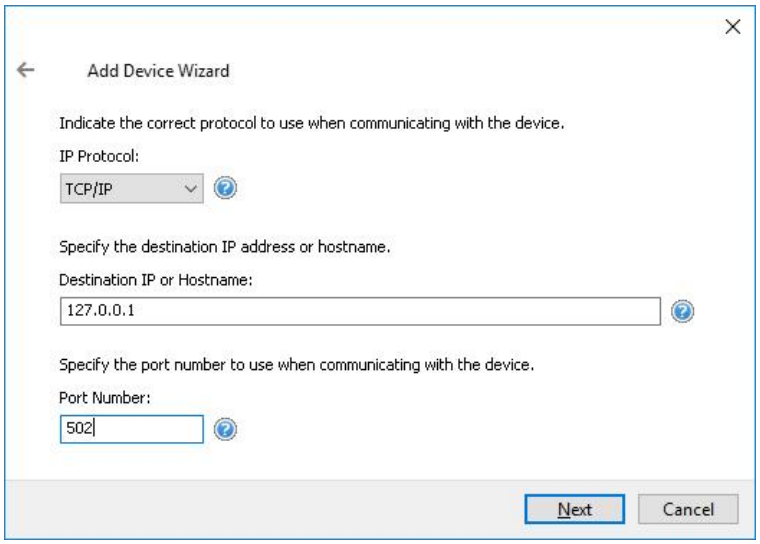
5. Leave all other values at the defaults, click **Next**, then **Finish**.

 ©2021-2025 PTC, Inc. All Rights Reserved.

3.8 Create a Modbus Ethernet Device

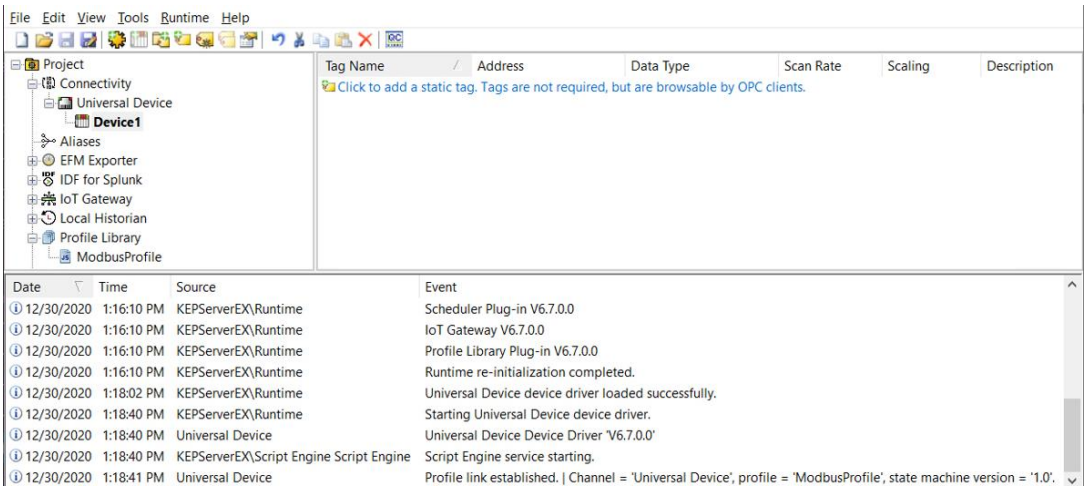
The next step is to configure a device under the new channel. This guide leaves most settings at the defaults, but updates the Destination IP or Hostname and the Port Number to connect to a Modbus server device (the steps would work for any Modbus device).

1. Select the new channel in the tree.
2. Right-click and choose **New Device...**.
3. Enter a name for the new device and click **Next**.
4. Accept or change the defaults, then click **Next**.
5. For the **Destination IP or Hostname** value, enter 127.0.0.1.
6. For the **Port Number**, enter 502.



- **Tip:** If you prefer to run this example with UDP as the underlying transport protocol please choose UDP for the IP Protocol property and set the Source- and Destination- IP/Port as described in the UDD help document.

7. Click **Next**, then **Finish**.



8. Verify that the following messages in the Event Log:

```
Script Engine service starting.
Profile link established. | Channel = 'Universal Device', profile =
'ModbusProfile', state machine version = '2.0', mode='Client'.
```

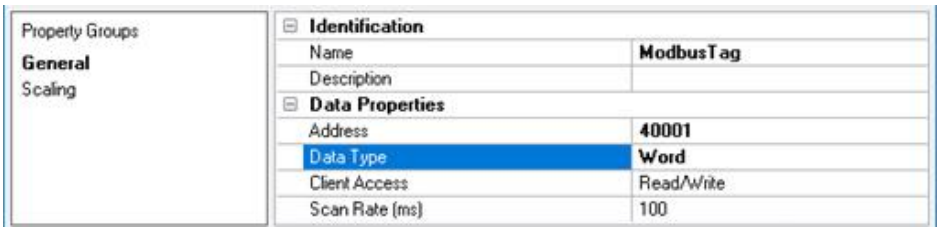
- **Tip**: Creating additional channels does not show "Script Engine Starting" message again unless all clients to Universal Device channels have been closed for at least five minutes. The "Profile link established..." message appears each time a channel successfully links to a profile. The link only attempts to establish after a device is created under a channel.

©2021-2025 PTC, Inc. All Rights Reserved.

3.9 Create Tags

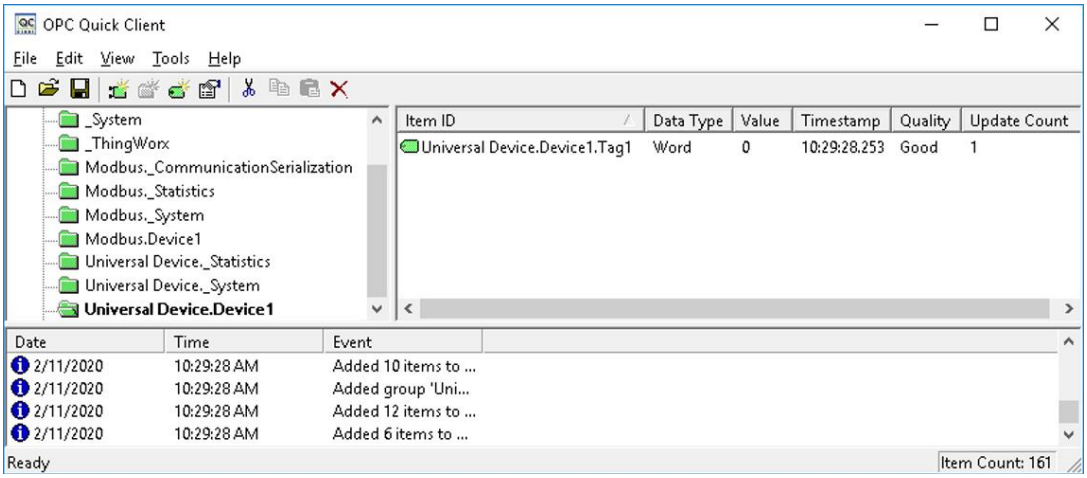
Once a device is defined, data can be collected through tags. A tag can be anything in the ranges expected by the profile script.

1. Select the new device in the tree.
2. Right-click and choose **New Tag**.
3. Enter a name and description for the new tag
4. Enter an Address of 40001 for this guide.
5. Select the Data Type **Word** from the drop-down and click **OK**.



3.10 Confirm Connectivity to Device

1. Launch OPC QuickClient from the configuration.



2. Verify the new tag for the universal device tag shows good quality data.
3. OPTIONAL: Update the Modbus device tag value (via QuickClient) and see that change reflected in the Universal Device tag. Do a write and verify the value update to the new value.

 ©2021-2025 PTC, Inc. All Rights Reserved.