kepware®

Connectivity Guide

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

February, 2021
Ref. 1.001

Table of Contents

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 TCP protocol. This is intended to help provide an understanding of the concepts of the Universal Device driver using Modbus TCP as an example and does not replace the Modbus features in KEPServerEX.

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 Kepware.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*

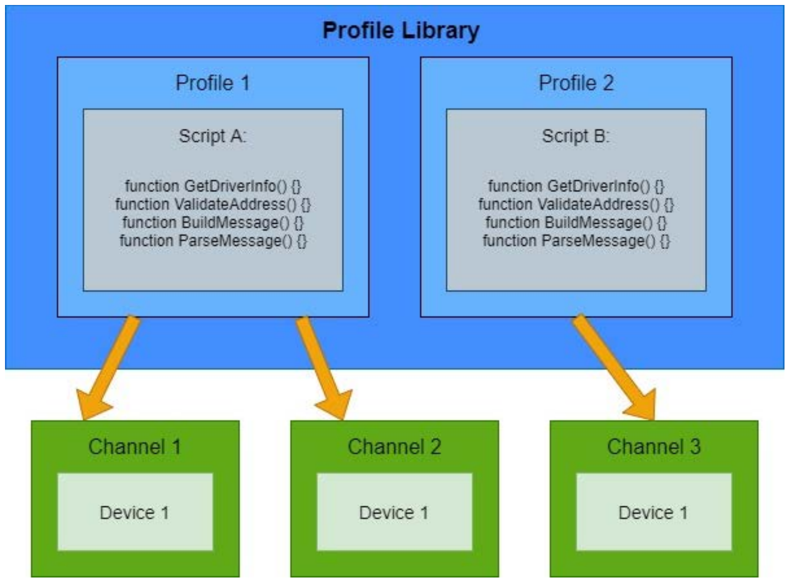
2. Key Concepts

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: GetDriverInfo, ValidateAddress, BuildMessage, and ParseMessage. These functions are described in further detail in the respective sections in this document. *See Script Function: GetDriverInfo, Script Function: ValidateAddress, Script Function: BuildMessage, and Script Function: ParseMessage.*

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.

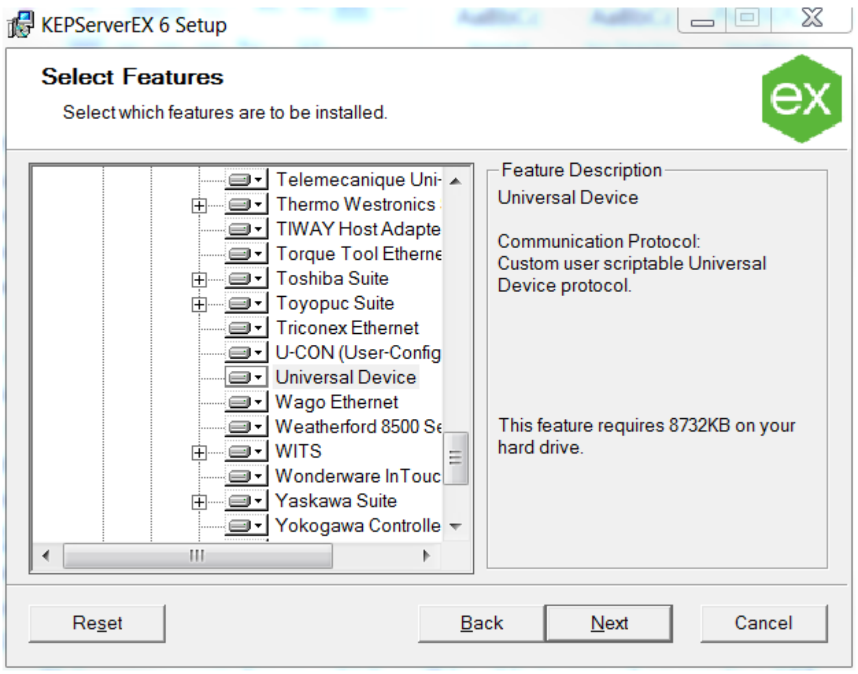


Profile Library
Profile 1
Script A:
function GetDriverInfo() {}
function ValidateAddress() {}
function BuildMessage() {}
function ParseMessage() {}
Profile 2
Script B:
function GetDriverInfo() {}
function ValidateAddress() {}
function BuildMessage() {}
function ParseMessage() {}
Channel 1
Device 1
Channel 2
Device 1
Channel 3
Device 1

3. Procedure

3.1 Installation

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

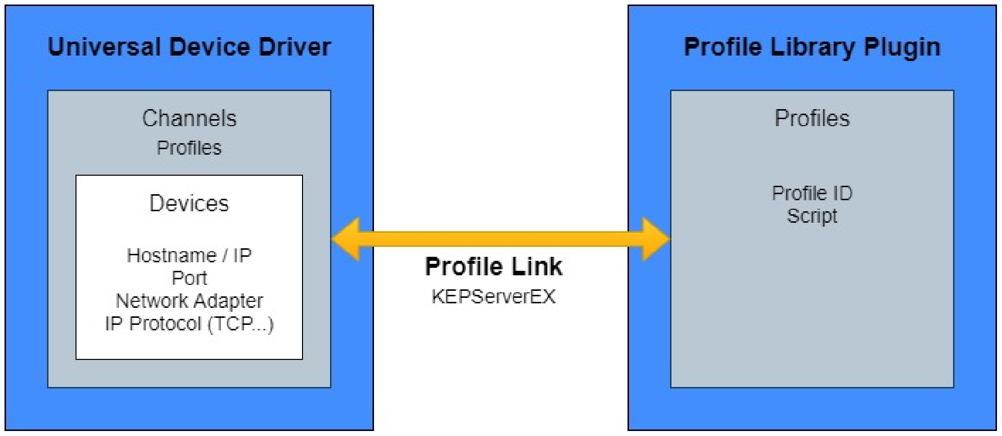


Feature Description
Universal Device

Communication Protocol:
Custom user scriptable Universal Device protocol.

This feature requires 8732KB on your hard drive.

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.

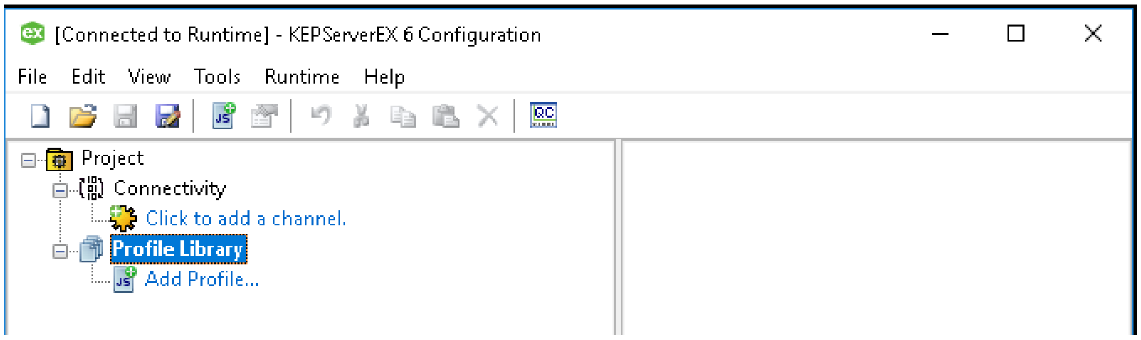
Profile ID: The next important piece of the profile is the Profile ID, formatted as an auto-generated GUID. The Profile ID is a unique identifier that the channel uses to establish a link to an existing Profile in the Profile Library. This allows for a project to have many channels linked to the same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

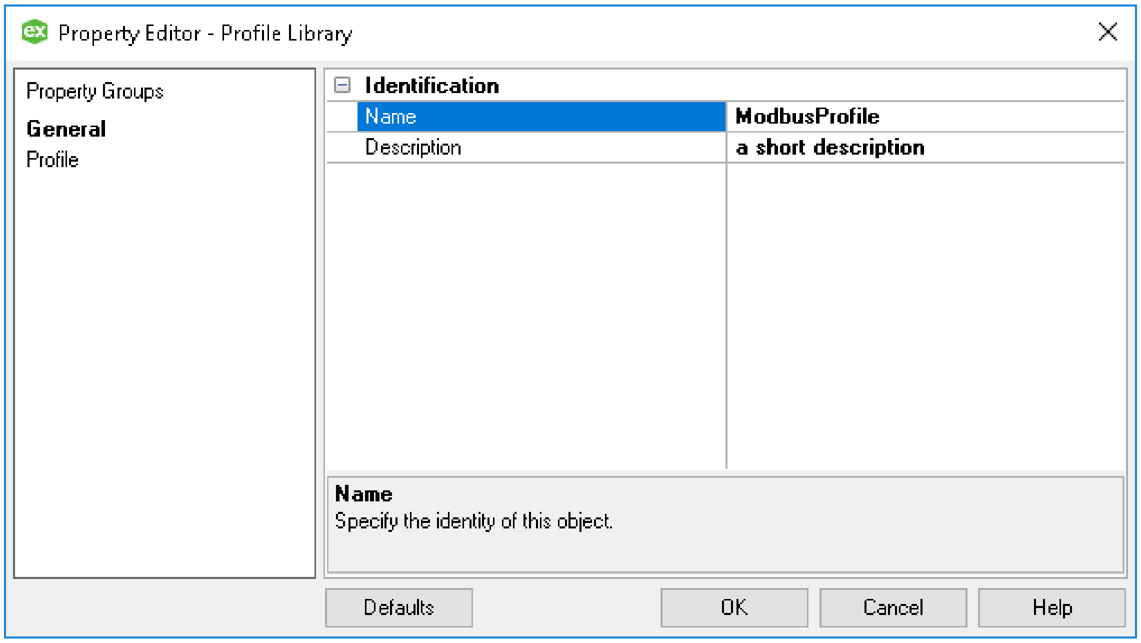
Description: A description of the profile.

3.3 Create a Profile with KEPServerEX 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....**


[Connected to Runtime] - KEPServerEX 6 Configuration
File Edit View Tools Runtime Help
Project
Connectivity
Click to add a channel.
Profile Library
Add Profile...

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


Property Editor - Profile Library

Property Groups
General
Profile

Identification	
Name	ModbusProfile
Description	a short description

Name
Specify the identity of this object.

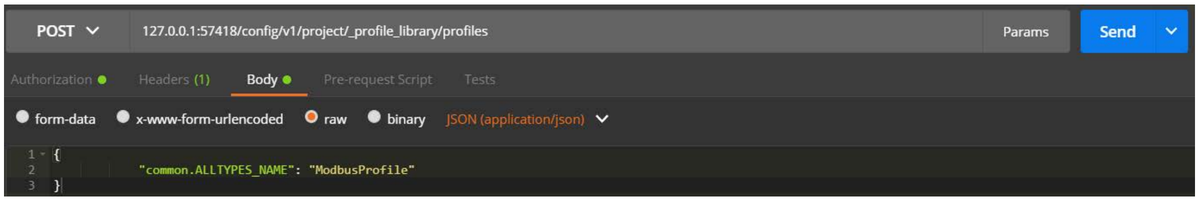
5. Locate the **Profile** property group.
6. Verify the Profile ID and default JavaScript profile script appear automatically.
7. Click **OK**. A profile has been created.

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.

1. 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:

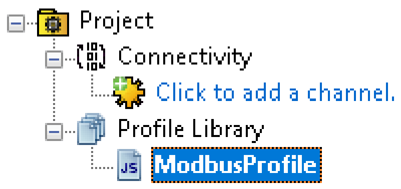
```
        {
                "common.ALLTYPES_NAME": "ModbusProfile"
        }
```

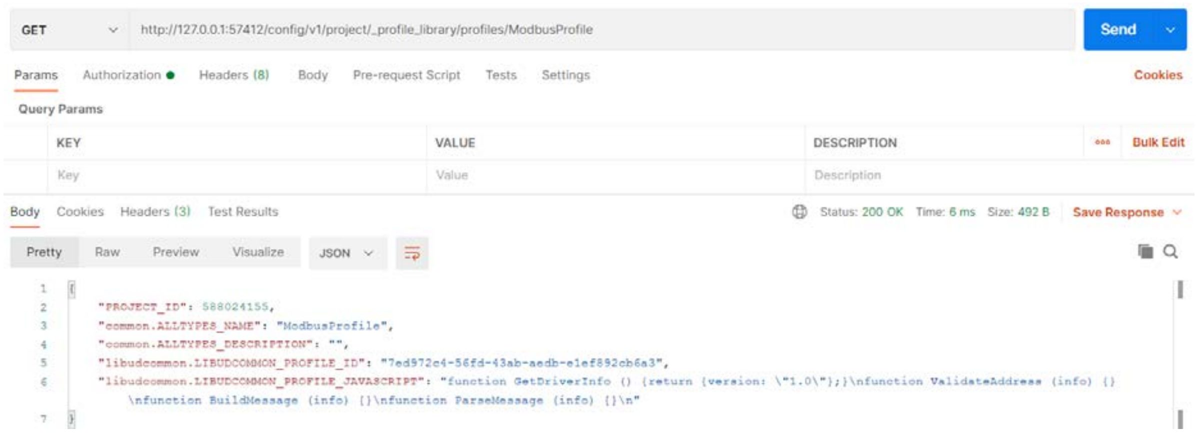

POST 127.0.0.1:57418/config/v1/project/_profile_library/profiles

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

Tip: This guide uses ModbusProfile as the profile name.

2. 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`.





http://127.0.0.1:57412/config/v1/project/_profile_library/profiles/ModbusProfile

Status: 200 OK Time: 6 ms Size: 492 B

```
{
    "PROJECT_ID": 588024155,
    "common.ALLTYPES_NAME": "ModbusProfile",
    "common.ALLTYPES_DESCRIPTION": "",
    "libudcommon.LIBUDCOMMON_PROFILE_ID": "7ed972c4-56fd-43ab-aedb-e1ef892cb6a3",
    "libudcommon.LIBUDCOMMON_PROFILE_JAVASCRIPT": "function GetDriverInfo () {return {version: \"1.0\"};}\nfunction ValidateAddress (info) {}
        \nfunction BuildMessage (info) {}\nfunction ParseMessage (info) {}\n"
}
```

3. 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.

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.
    - **GetDriverInfo**: Retrieves driver metadata
    - **ValidateAddress**: 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
    - **BuildMessage**: Builds a frame of bytes to transmit to the device across the wire.
    - **ParseMessage**: Interprets the response from the device and updates tag values or indicates is the read or write operation was successful based on the data in the response.
3. Build out the script one function at a time, use the following information to edit the script.

    The **GetDriverInfo** 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 UDD with which it was created.

    Note: The only supported version is 1.0. 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:

    - GetDriverInfo causes all subsequent operations to fail until corrected
    - ValidateAddress causes the tag address to be treated as "invalid"
    - BuildMessage causes the read or write operation on the current tag to fail
    - ParseMessage causes the read or write operation on the current tag to fail

Below is the entire GetDriverInfo function:

```
function GetDriverInfo() {
    return { version: “1.0” };
}
```

The **ValidateAddress** 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.

If desired, add logic to this function to modify various tag fields, such as providing a valid default data type or modify an address format to enforce consistency among tag addresses.

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

```
// Validate the address is a holding register in the supported range
let tagAddress = info.tag.address;
try {
    let numericAddress = parseInt(tagAddress, 10);
    if ((numericAddress < 40001) || numericAddress > 49999 || isNaN(numericAddress)) {
        info.tag.valid = false;
        return info.tag;
    }

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

The code above offers a look at the JavaScript object `info` that the driver provides to the script writer. This object is meant to hold data to be exchanged between the script and the driver. It checks the address received from the driver (`info.tag.address`) and verifies it is in the expected range for a Modbus holding register. If it’s not in that range, fail the tag being added by setting the valid field of the tag to false: `info.tag.valid = false`.

```
// Provide a valid default data type based on register
// Note:  "Default" is an invalid data type
let validDataTypes = {"3": "Word", "4": "Word"}
if (info.tag.dataType === "Default") {
    let registerChar = info.tag.address.charAt(0);
    info.tag.dataType = validDataTypes[registerChar];
}
```

```
/*
 * The regular expression to compare address to.
 * ^4   starts with '4'
 * 0*   find zero or more occurrences of '0'
 * 1$   ends with '1'
 */
let addressRegex = /^40*1$/;

// Correct a "semi-correct" tag address (e.g. 401 or 400001 --> 40001) with regex
if (addressRegex.test(info.tag.address)) {
    info.tag.address = "40001";
}
```

The above code provides examples of logic to modify various tag fields. The first code block resets the data type if Default is initially selected. While Default is a KEPServerEx 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 based on the register if the data type is set as Default.

The second code block uses a regex to recognize semi-correct addresses and modify them accordingly. In the above implementation, this logic adjusts tag addresses with too few or too many zeros; for example, '401' or '400001` is changed to '40001'.

Below is the entire ValidateAddress function:

```
function ValidateAddress(info) {
// Provide a valid default data type based on register
// Note:  "Default" is an invalid data type
let validDataTypes = {"3": "Word", "4": "Word"}
if (info.tag.dataType === "Default") {
    let registerChar = info.tag.address.charAt(0);
    info.tag.dataType = validDataTypes[registerChar];
}

/*
 * The regular expression to compare address to.
 * ^4   starts with '4'
 * 0*   find zero or more occurrences of '0'
 * 1$   ends with '1'
 */
let addressRegex = /^40*1$/;

// Correct a "semi-correct" tag address (e.g. 401 or 400001 --> 40001) with regex
if (addressRegex.test(info.tag.address)) {
    info.tag.address = "40001";
}

// Validate the address is a holding register in the supported range
let tagAddress = info.tag.address;
try {
let numericAddress = parseInt(tagAddress, 10);
if ((numericAddress < 40001) || numericAddress > 49999 || isNaN(numericAddress)) {
        info.tag.valid = false;
        return info.tag;
    }

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

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

```
function BuildMessage(info) {
let buildStatus = "Failure";

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

    // Evaluate if the frame was successfully built
    if (readFrame.length === 12) {
        buildStatus = "Receive";
    }

    return { status: buildStatus, data: readFrame };

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

    // Evaluate if the frame was successfully built
    if (SENTWRITEFRAME.length === 12) {
        buildStatus = "Receive";
    }

    return { status: buildStatus, data: SENTWRITEFRAME };
}
}
```

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

Helper Function: BuildReadMessage

This function builds into a frame 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.

Note: While the Modbus protocol supports blocking / bulk read and write functionality, version 1.0 of the Universal Device Driver does not. Only one tag can be processed at a time. As such, the tags parameter is expected to be an array containing a single tag element.

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

// 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
let first = registers[0];
let count = registers[registers.length - 1] - first + 1;
// Get the zero-based register index to make the request
first -= 40001;
```

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

```
// 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 that its building frames 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's shared between the script and driver and means that it 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 frame
let frame =
// ----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 frame being constructed. It is an array of bytes to be sent to the Modbus device. The code comments the different parts of the frame 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 frame 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.";
}

// 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
let first = registers[0];
let count = registers[registers.length - 1] - first + 1;
// Get the zero-based register index to make the request
first -= 40001;

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

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

---|-------Register count--------|
loByte(first), hiByte(count), loByte(count)]

return frame;
}
```

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
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 -= 40001;

// Get the value to write which is located in the first
// element in the tags[n].value object
let value = parseInt(tags[0].value);
```

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.

```
// Build the Modbus Ethernet frame
let frame =
   // ----Transaction ID-----|-Protocol--|---Length--|Server|-Fxn-|

    [
    hiByte(TXID), loByte(TXID), 0x00, 0x00, 0x00, 0x06, 0x00, 0x06,
       --------Starting Address----------|-------value to write--------|
       hiByte(register), loByte(register), hiByte(value), loByte(value)
   ];

return frame;
```

The above shows how to build up a write frame, which is very similar to building a read frame within the BuildReadMessage function.

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

```
function ParseMessage(info) {
let parseStatus = "Failure";

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) {
        parseStatus = "Success";
    }

    return { status: parseStatus, tags: tags };

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

Helper Function: ParseReadMessage

This function's purpose is to parse an incoming frame into a tag value to update the respective tag in the server. The incoming frame 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.

Note: As noted previously, version 1.0 of the Universal Device Driver does not support blocking / bulk reads or writes. Only a single tag can be processed at a time.

```
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.";
}

// Convert the string addresses to integers (e.g. 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);

// MBE Response values start here:
let offset = 9;
// Enough bytes?
if (data.length < offset + 2 * registers.length) {
    throw "Invalid response from device";
}
```

The code above performs error checking and getting some information about the transaction.

```
// Iterate the registers and lookup the response value for each
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
    let hi = data[2*index + offset];
    let lo = data[2*index + offset + 1];
    tags[i].value = (wordFromBytes (hi, lo));
}

return tags;
```

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.";
}

// Convert the string addresses to integers (e.g. 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);

// MBE Response values start here:
let offset = 9;
// Enough bytes?
if (data.length < offset + 2 * registers.length) {
    throw "Invalid response from device";
}

// Iterate the registers and lookup the response value for each
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 it from the response buffer
    let hi = data[2*index + offset];
    let lo = data[2*index + offset + 1];
    tags[i].value = (wordFromBytes (hi, lo));
}

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 SENTWRITEFRAME.

Below is the entire ParseWriteMessage function:

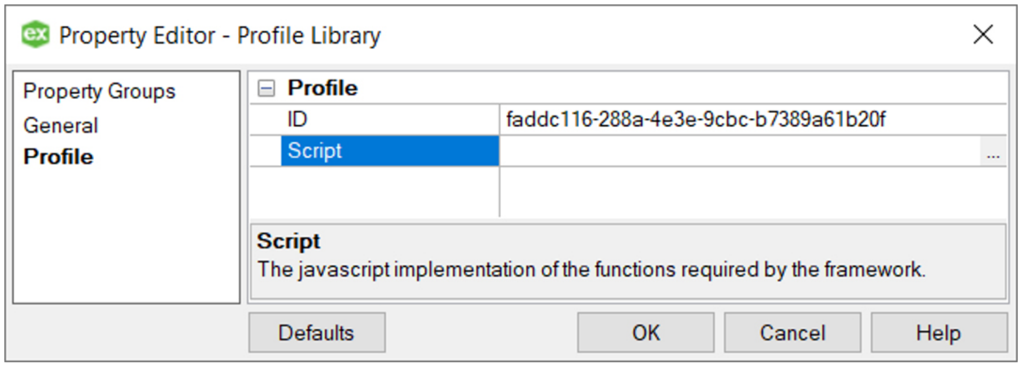
```
function ParseWriteMessage(data) {
// Modbus echoes a write request so if the frame sent
// does not match the frame received, then the write fails
SENTWRITEFRAME.forEach((e1) => data.forEach((e2) => {
    if (e1 !== e2) {
        return "Failure";
    }
}));
return "Success";
}
```

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**.



Profile

ID	faddc116-288a-4e3e-9cbc-b7389a61b20f
Script	

Script
The javascript implementation of the functions required by the framework.

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**.

3.6.2 Using the Configuration API

1. Create a PUT to the profile at URL

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

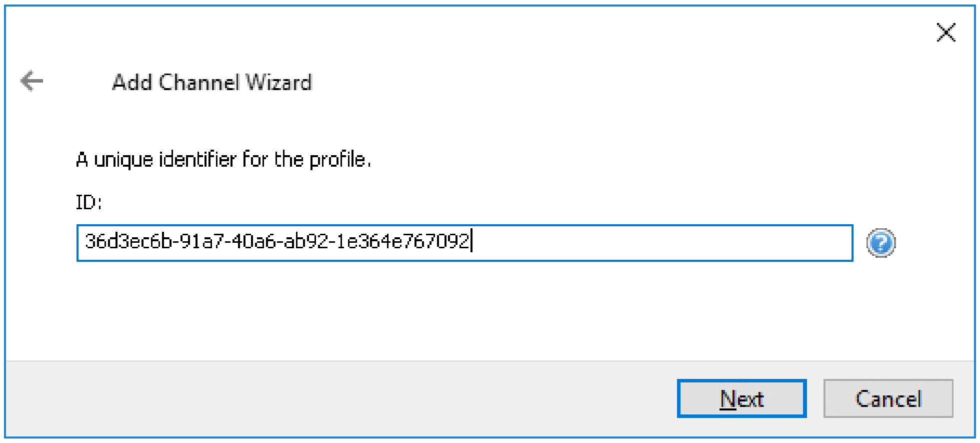
   with body:
   ```
   {
         "FORCE_UPDATE": true,
         "libudcommon.LIBUDCOMMON_PROFILE_JAVASCRIPT": <ESCAPED SCRIPT>
   }
   ```

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.

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).


Add Channel Wizard

A unique identifier for the profile.

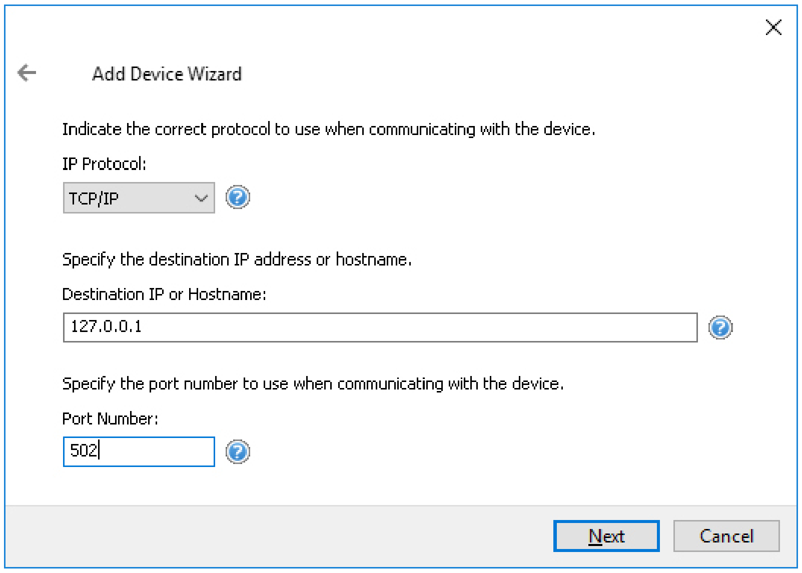
ID: 36d3ec6b-91a7-40a6-ab92-1e364e767092

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

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.


Add Device Wizard

Indicate the correct protocol to use when communicating with the device.

IP Protocol: TCP/IP

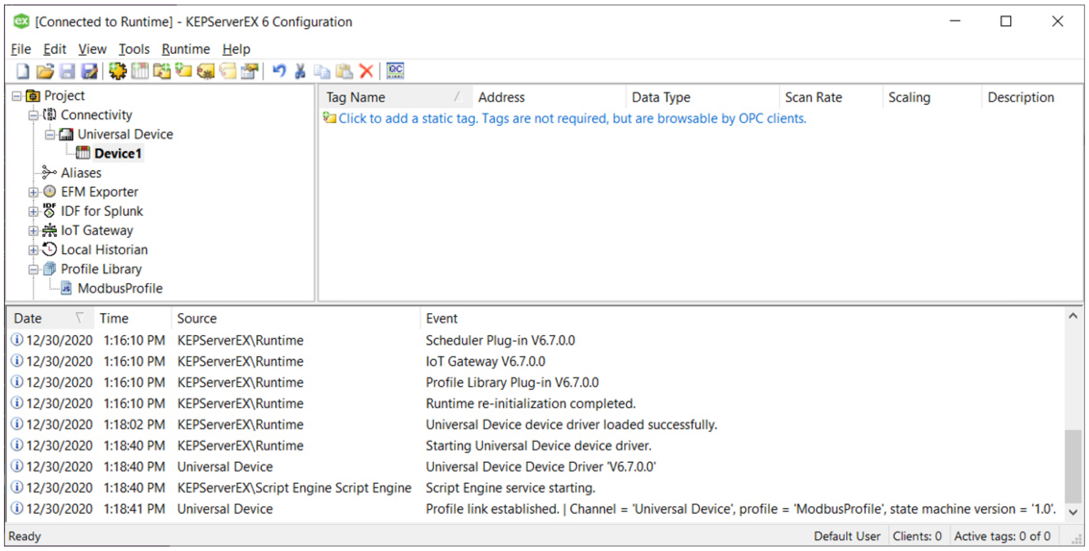
Specify the destination IP address or hostname.

Destination IP or Hostname: 127.0.0.1

Specify the port number to use when communicating with the device.

Port Number: 502

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


[Connected to Runtime] - KEPServerEX 6 Configuration

Click to add a static tag. Tags are not required, but are browsable by OPC clients.

```
Date        Time        Source                                 Event
12/30/2020  1:16:10 PM  KEPServerEX\Runtime                    Scheduler Plug-in V6.7.0.0
12/30/2020  1:16:10 PM  KEPServerEX\Runtime                    IoT Gateway V6.7.0.0
12/30/2020  1:16:10 PM  KEPServerEX\Runtime                    Profile Library Plug-in V6.7.0.0
12/30/2020  1:16:10 PM  KEPServerEX\Runtime                    Runtime re-initialization completed.
12/30/2020  1:18:02 PM  KEPServerEX\Runtime                    Universal Device device driver loaded successfully.
12/30/2020  1:18:40 PM  KEPServerEX\Runtime                    Starting Universal Device device driver.
12/30/2020  1:18:40 PM  Universal Device                       Universal Device Device Driver 'V6.7.0.0'
12/30/2020  1:18:40 PM  KEPServerEX\Script Engine Script Engine Script Engine service starting.
12/30/2020  1:18:41 PM  Universal Device                       Profile link established. | Channel = 'Universal Device', profile = 'ModbusProfile', state machine version = '1.0'.
```

Ready   Default User   Clients: 0   Active tags: 0 of 0

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

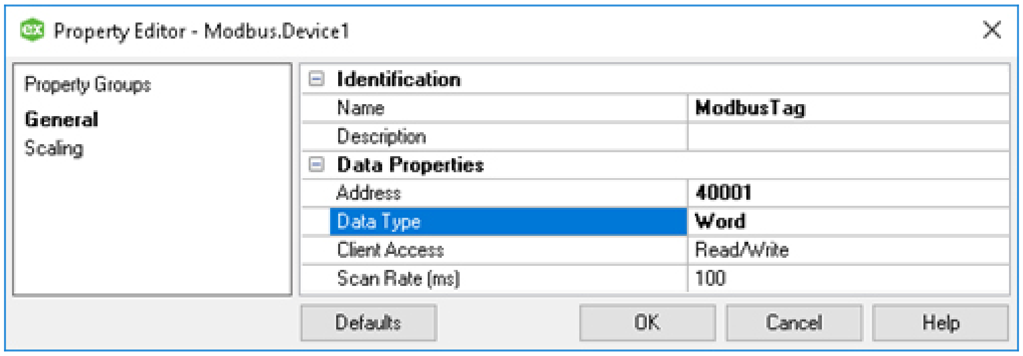
```
Script Engine service starting.
Profile link established. | Channel = ‘Universal Device’, profile =
‘ModbusProfile’, state machine version = ‘1.0’.
```

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 with a profile. The link only attempts to establish after a device is created under a channel.

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.

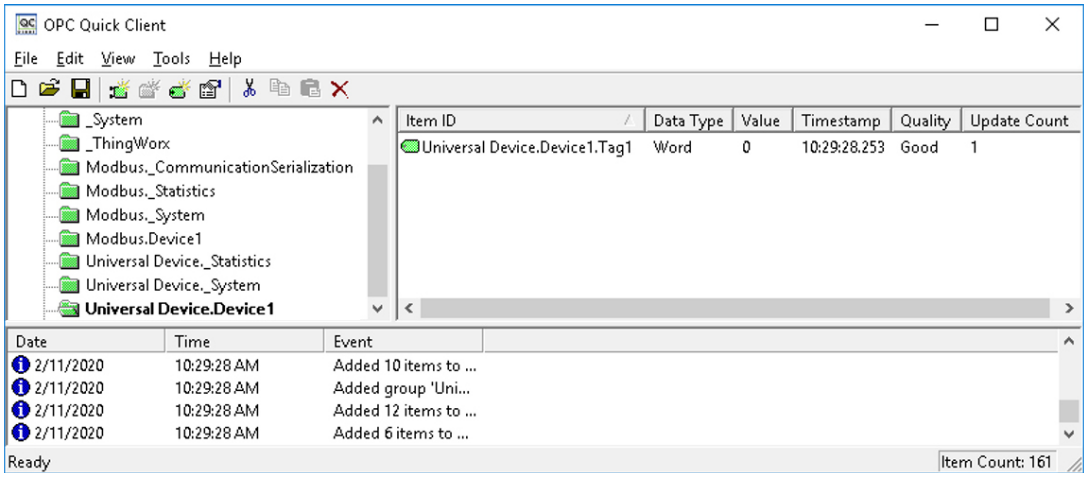
1. Select the new device in the tree.
2. Right-click and choose **New Tag**.
3. Enter a name and description for the new tag
4. Enter an Address of 40001 for this guide.
5. Select the Data Type **Word** from the drop-down and click **OK**.



Identification	
Name	**ModbusTag**
Description	
Data Properties	
Address	**40001**
Data Type	**Word**
Client Access	Read/Write
Scan Rate (ms)	100

3.10 Confirm Connectivity to Device

1. Launch OPC QuickClient from the configuration.



Item ID	Data Type	Value	Timestamp	Quality	Update Count
Universal Device.Device1.Tag1	Word	0	10:29:28.253	Good	1

Date	Time	Event
2/11/2020	10:29:28 AM	Added 10 items to ...
2/11/2020	10:29:28 AM	Added group 'Uni...
2/11/2020	10:29:28 AM	Added 12 items to ...
2/11/2020	10:29:28 AM	Added 6 items to ...

2. Verify the new tag for the universal device tag shows good quality data.
3. OPTIONAL: Update the 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.