Profile Library Plug-In

Table of Contents

Profile Library Plug-In

Help version 1.015

CONTENTS

Overview

The Profile Library Plug-In allows a user to create script-based profiles to use in conjunction with the Universal Device Driver to communicate with a wide variety of Ethernet devices. Make use of the Profile Library Plug-In to implement custom profiles in cases when there is no native driver for a particular protocol or device. The Profile Library Plug-In offers the following features:

- Ability to customize profiles to meet specific connectivity needs
- Script-based interface which gives users flexibility in what functionality to implement
- Ability to edit a profile once and push edits to all instances of that profile

Architecture

The Profile Library Plug-in is used to configure and maintain profiles that are consumed by the Universal Device Driver. These profiles contain all the information and instructions (through script, profile properties, and profile parameters) necessary to communicate with a device. The script defines the interface and implementation required to build and handles frames of a specific protocol that your device communicates over.

There is a one-to-one relationship between Universal Device Driver channels and profiles, and a profile must be assigned to a channel upon creation. This process is called linking the profile to the channel. Once a profile is linked and a device is created on the channel, the profile’s script is registered with the server's script engine, which prepares the server and driver for communication using the profile’s script. At this point all communication happens between the script engine, driver, and device.

If the profile is updated or modified the active script will be un-registered and the updated script from the modified profile will be registered with the script engine. Once this process is complete, all driver communication uses the updated script. This same process applies if other properties on the profile are created, modified, or deleted as well, such as profile parameter definitions.

Creating and Configuring Profiles

The Profile Library Plug-In provides the ability to create profiles via the server configuration interface and the Configuration API Service.

Note: For communication to occur, a channel must be linked to a valid profile.

What is a profile?

A Profile is a collection of properties that together provide all the information that the Universal Device Driver needs to communicate with a device. Properties on the profile include the Name, Description, ID, and Script. Of these the script and ID are the most important. The script here defines all the instructions required by the driver to communicate over a specific protocol. The script interface is defined in more depth in the script section.

Tip: The profile's ID property is a unique identifier that is used to link the profile to a Universal Device Driver channel and is in a GUID format.

Profile Properties — General

These profile properties are specific to the Profile Library Plug-In and are associated with every profile.

- **Name**: This property specifies a name for the profile.
- **Description**: User-defined information about this profile.

Profile Properties — Profiles

These profile properties are specific to the Profile Library Plug-In and are associated with every profile. Default values are automatically generated for each property.

- **ID**: This property is a unique identifier in the form of a GUID which links Universal Device Driver channels to profiles defined in the Profile Library Plug-In.
- **Script**: This property is the JavaScript code that communicates with a device and provides the Universal Device Driver with the information it needs to communicate to the device over its protocol. If using the configuration UI, upload a JavaScript file by clicking on the browse ellipses (...) on the right side of the text box. If changes are made to the script, the file must be uploaded again. This method of file upload supports UTF-8 encoded files. If there are any channels linked to a profile, it is necessary to reinitialize the server after uploading the script.

Creating a Profile Script

The profile script is a property of the profile that contains the JavaScript to execute functions required to validate tags and communicate with a device. The script can be created and edited offline in any editor.

Ethernet Solicited

The only type of profile currently supported is an Ethernet solicited profile using TCP/IP. This is a type of communication with a device that needs to be asked for data by the server. The Ethernet solicited profile script has several required functions: GetDriverInfo, ValidateAddress, BuildMessage, and ParseMessage. The purpose of these functions is described below.

Required Functions

The script is a collection of functions called and executed when needed. The user can add as many extra functions as desired to simplify complex operations. To perform basic I/O operations with a device, the functions described in this section must be included.

GetDriverInfo
The **GetDriverInfo** function identifies the interface contract between the script and the driver.

Input
None

Output
GetDriverInfo must return a JavaScript object with the following fields:

- version – (String) Version string, with format <Major.Minor> (for example, “1.0”).

  Note: The only currently supported version is “1.0”. Any other value is rejected by the driver, leading to failure of all subsequent script functions.

ValidateAddress
The purpose of the ValidateAddress function is to specify and validate the address syntax of a tag, which is passed into the function via an object argument, to be used when adding tags and ultimately parsing tag addresses when communicating with a device. The script writer also provide logic to "correct" or modify a tag address (tag.address) if necessary. For instance, it can change the format slightly to enforce consistency among tag addresses, such as ‘k01’ adjusted to ‘k0001’. The ValidateAddress function also verifies and "corrects" the datatype (tag.dataType) and read only (tag.readOnly) tag properties.

Input
ValidateAddress has a single input argument. The argument is a JavaScript object with the following fields:

- tag – (object) Represents the tag to be validated. It has the following fields:
  - address – (String) Tag address.
  - dataType – (String) server data type *(see **Data Type** section for valid values)*.
  - readOnly – (Bool) true for read only, false for read/write tags.

Output
ValidateAddress must return a JavaScript object with the following fields:

- address – (String) Tag address. Can be modified if necessary, e.g. expanding ‘k01’ to ‘k0001’ (optional).
- dataType – (String) server data type, modified if necessary *(see **Data Type** section for valid values)* (optional).
  Note: A return value of Default is invalid; a data type must be specified if the input value is Default (optional).
- readOnly – (Bool) true for read only, false for read/write tags, modified if necessary (optional).
- valid – (Bool) true if the tag is valid, false if the tag is invalid (required).

BuildMessage
The Universal Device Driver communicates with devices, whether to read or write to a register, with the BuildMessage function. The function receives a JavaScript object that contains the type of message to build and a one-element array of tag objects. The tag object contains the tag address, datatype, value, and a Boolean representing whether the tag is read-only or read-write. The type of message can be Read or Write. It is also possible to create a conditional message (for example, an IF statement) based on what type of message to be built.

The expected output for this function is the status and data. The status field informs the state machine within the Universal Device Driver of the next action. If a problem or error is encountered with this JavaScript object or script, it returns a status of Failure. If a status of failure is reported, the data array is not required. A status of Receive means the driver enters a receiving state and waits for the device to respond with data. Once the data is received, it is passed to the ParseMessage function. Along with status, BuildMessage returns the array of bytes to be sent to the device.

Input
BuildMessage has a single input argument. The argument is a JavaScript object with the following fields:

- type – (String) the context in which this method is called. The value is either Read or Write.
- tags – (Array of tag objects) The array of tags being read or written. The tag object has the following fields:
    - address – (String) tag address
    - value – (variant) desired value of the tag. This field only exists when type is Write.
    - dataType – (String) server data type *(see* ***Data Type*** *section for valid values)*.
    - readOnly – (Bool) true for read only; false for read/write tags.

Output
BuildMessage must return a JavaScript object with the following fields:

- status – (String) This field is used to determine the driver's next action. Valid return status is Receive or Failure. This is the only field required when the type is Write.
    - Receive – Returning a status of Receive tells the driver that you are expecting a response. The bytes the driver receives are passed into the ParseMessage function on a subsequent call.
    - Failure – Returning a status of Failure tells the driver that an error has been encountered and the read or write failed. ParseMessage is not called in this case.
- data – (int array) The packet of data to send to the device. This field is only required when status is not Failure.

ParseMessage
When the Universal Device Driver receives a response from the device, the ParseMessage function reads and parses the data. Just like BuildMessage, the types of message parsed are either read or write. The data to parse and the tag object are also included in the input argument. Using this information, set the new value of the tag or fail the transaction, which results in the tag having bad quality.

Input
ParseMessage has a single input argument. The argument is a JavaScript object with the following fields:

- type – (String) the context in which this method is called. The value is either Read or Write.
- data – (Array) the packet of data sent from the device to the Universal Device Driver.
- tags – (Array of tag objects) The array of tags that is being read or written. The Tag object has the following fields:
    - address – (String) tag address.
    - value – (Variant) tag value that was written. This field only exists when type is Write.
    - dataType – (String) server data type *(see* ***Data Type*** *section for valid values)*.
    - readOnly – (Bool) true for read only; false for read/write tags.

Output
ParseMessage must return a JavaScript object with the following fields:

- status – (String) This field is used to determine what the driver does next. Valid return status is Success or Failure.
  - Success – returning a status of Success means the transaction between the driver and device successfully completed with no errors.
  - Failure – returning a status of Failure means an error was encountered and the read or write failed.
- tags – (array of tag objects) This tag field is required when type is Read and status is not Failure. This tag object must contain the following fields:
  - address – (String) tag address
  - value – (variant) the parsed value read from the packet.

See Also: *View an Existing Profile*

Utility Functions

In addition to the required functions, a user can create as many additional functions as desired to simplify complex operations. The sample profiles provide examples of helper functions that can assist with converting data types or determining the validity of the data returned from a device.

There are also native JavaScript functions that can assist users in creating and debugging a script. For example, users can log messages to the Event Log through the log() function. When an error is encountered, it is considered best practice to log a message with helpful information about what happened and return a status of failure rather than throwing an exception.

Additionally, the log() function can be used to determine if the expected data is being created and passed between functions correctly. Since the required methods make use of JavaScript objects, it is helpful to pair the log() function with the JSON.stringify() function. This function converts a JavaScript object to a string that makes it more human-readable in the Event Log.

Additional Script Information

The script is a collection of functions called and executed when needed. The user can add as many extra functions as desired to simplify complex operations.
See Also: *Profile Library Modbus Tutorial (contact support)*

Data Type

The dataType strings used by the profile are derived from the data types supported by the server. The valid values for dataType are:

- Default
  Note: This is not a valid return value; Default will only appear as an input.
- String
- Boolean
- Char
- Byte
- Short
- Word
- Long
- DWord
- Float
- Double

- BCD
- LBCD
- LLong
- QWord

Script Writing Best Practices

- Use short meaningful variable and function names:
  - String xyz; // this variable name does not describe what it is
  - String tagAddress; // this variable is clearly used to hold the tag address value
- Comment thoroughly and remember that good code explains itself
- Create functions that do one thing
  - ConvertStringToByteArrayAndCreateMessage(){} // This function is responsible for too many tasks. This also makes the function hard to read.
- Avoid using Global variables as much as possible; use local variables instead.
- Be careful of while loops. If while loops are done improperly, they can loop forever and the server will timeout and fail the operation.

Adding the Script to the Profile

To add the script to the profile, it must be included in the body of a POST on profile creation or in the body of a PUT during a profile update or added via the thick client. When using the thick client, an entire script can be pasted in the test box. However, it is necessary to use Linux line endings or the entire script will not paste into the script property text box. It is recommend using a JavaScript IDE or text editor to create and modify the script, then copy and paste the script into the profile script property or browse to locate the script.

Using the Configuration API

This section describes the process to create a profile using the Configuration API Service. The steps shown here can be used to create any of the profile types. For a template of functions required for a specific profile type, create a new profile with no script defined in the request body, then send a GET API request.
Tip: This documentation assumes the user is on the same machine as the server and is using the default HTTP port. Therefore, localhost:57412 is used as the address for all API calls. Change the IP address and port as needed.

Sending API Requests

The API is accessible through a REST interface that can act on HTTP requests.
See the Configuration API Service help documentation in server help under Configuration API Service section for more information about interfacing with the server over the API.

Creating a Profile

To create a profile using the API, send a POST to the following endpoint:

```
POST http://localhost:57412/config/v1/project/_profile_library/profiles
```

with a body:

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

If the profile is created with only its name defined in the POST request, the server populates the Script and ProfileID fields. The Script field then contains a template script that can be retrieved with a GET request (see View an existing Profile) as a starting point.

The user can optionally specify a description; the JavaScript that makes up the “driver logic” and a reference ID in the form of a UUID. The body of a POST including these properties should look like:

```
{
"common.ALLTYPES_NAME": "Profile_Name_Here"
"common.ALLTYPES_DESCRIPTION": "description_here",
"libudcommon.LIBUDCOMMON_PROFILE_JAVASCRIPT": "<javascript>",
"libudcommon.LIBUDCOMMON_PROFILE_ID": "<UUID>"
}
```

Updating a Profile

To update a profile, send a PUT request to the endpoint, and append "/profiles/" and the profile name, in the form of:

```
PUT http://localhost:57412/config/v1/project/_profile_library/profiles/<profile_name>
```

It is not recommended to update profiles with an active client reference. When a linked profile is updated, Tags on any linked channels will report "bad quality" until the new script or profile configuration is propagated to each of the linked channels. At that point, assuming that the profile is valid and works correctly with the existing channel configurations, those tags will restart communication and begin reporting ‘good’ data again.

It is possible that updating a profile will cause linked channels to become invalid. For example, if the ValidateAddress function changes and your static tag, or dynamic client tag, addresses no longer fit the address schema in the new function, those tags that no longer pass validation will remain in ‘bad quality’ until the profile and or link is updated or modified again.

Tip: Once a profile is updated, reinitialize to apply the changes.

View an Existing Profile

To view the contents in an existing profile, send a GET request to the endpoint and append "/profiles/" and the profile name, in the form of:

```
GET http://localhost:57412/config/v1/project/_profile_library/profiles/<profile_name>
```

Note: If some of the properties of the profile were generated by the server (properties that were omitted from the POST request to create the profile), they can be viewed in the GET response.

Index