Profile Library Plug-In

Table of Contents

Profile Library Plug-In

Help version 1.024

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 a script that implements business logic for device communication. The script defines the interface and implementation required for a specific protocol that the device uses to communicate.

Universal Device Driver channels require a linked profile that must be assigned during channel 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 is un-registered and the updated script from the modified profile registered with the script engine. Once this process is complete, all driver communication uses the updated script.

Important Concepts

- **Client / Server**: This references the type of Ethernet connection established with a device. In client mode, a TCP/IP socket connection is initiated by the Universal Device Driver to the specified IP

address using the defined port number. In server mode, the Universal Device Driver listens for an incoming connection on the specified IP address and port number.

- **Solicited**: This refers to a communication mode in which the Universal Device Driver requests data from a device.
- **Unsolicited**: This refers to a communication mode in which data is sent to the Universal Device Driver from a device without any corresponding request.
- **Data Cache**: A built-in cache library is provided to enable asynchronous communications, allowing the script to internally store tag data until it is needed for a read or write operation.
- **Transaction**: A discrete set of I/O operations with a device to perform some action, such as reading or writing data, starting a subscription, or completing a handshake.
- **Completing a Tag**: Tags can be Read or Write and need to be “completed” by the driver. When tags are requested, it is the responsibility of the script to perform the necessary steps to complete that request. Only when active tags are completed (successfully or unsuccessfully) are the next tags requested.

See Also: Profile Types

Profile Types

The Universal Device Driver provides a flexible framework for implementing communication protocols. Most protocol types fall into one of the following four broad categories, independent of the underlying transport mechanism. Each category provides general design guidance, but choosing one does not restrict its functionality.

- **Simple Solicited**: In Simple Solicited, tags are fulfilled using synchronous data requests as needed. No messages are expected from the device when there is no outstanding request.
- **Simple Unsolicited**: In Simple Unsolicited, the script never requests data. The device determines what data to send and when to send it. Tags are fulfilled from a cache populated by the script as data becomes available.
- **Mixed Mode**: In Mixed Mode, some tags are fulfilled synchronously and others populated asynchronously. The script determines that based on user-defined criteria for how to process each tag request.
- **Pub / Sub**: In Pub / Sub, tags are fulfilled asynchronously only after a request is made to the device that enables data publication. The script must have a mechanism for tracking if a subscription already exists for tags as they are requested.

Tip: Examples and templates can be found in the install directory \<product>\Examples\Universal Device Sample Profiles and at https://github.com/PTCInc/Universal-Device-Driver-Examples.

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 contains the JavaScript code that implements the required event handlers and business logic for the Universal Device Driver to communicate with a device. To load a script file, select the browse ellipses (...) on the right side of the text box and choose a JavaScript source file from disk.

  Notes:
  - Script files must be UTF-8 encoded and can be created and edited offline in any editor.
  - If changes are made to the script, the file must be uploaded again.
  - 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.

Required Functions

For Universal Device Driver to perform basic IO operations with a device the script must contain the following functions to handle events. The user can add additional functions, classes, variables, and global state as desired to simplify complex operations.

- onProfileLoad
- onValidateTag
- onTagsRequest
- onData

onProfileLoad
The **onProfileLoad** function defines the interface contract between the profile and the driver.

Input
None

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

- version – (string) Version string, with format <Major.Minor> (for example, “2.0”).
  Note: The only currently supported version is “2.0”. Any other value is rejected by the driver, leading to failure of all subsequent script functions.
- mode – (string) The communication mode of the port. Valid values are Client and Server. In Client mode, the Universal Device Driver acts as a client and opens a connection to the device. In Server mode, the Universal Device Driver acts as a server and the port is set to listen. This field is optional and defaults to Client.

onValidateTag
This function is called when the address, data type, and read-only mode of a tag must be validated. The function can correct or modify a tag address, data type and read-only mode if necessary. For example, it can change the format slightly to enforce consistency among tag addresses, such as ‘k01’ adjusted to ‘k0001’. Similarly, the onValidateTag function can correct or modify the data type and read-only tag properties. Additionally, this function can assign a bulk ID to each tag. The Universal Device Driver groups the tags with the same bulk ID together when providing tags to the onTagsRequest or onData functions.

Input
onValidateTag 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 Types** section for valid values)*.
  - readOnly – (Boolean) true for read only, false for read/write tags.

Output
onValidateTag 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 Types** 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.
- readOnly – (Boolean) true for read only, false for read/write tags, modified if necessary (optional).
- bulkId – (number) Integer value that identifies the group into which to bulk the tag with other tags. If no bulkId is provided, the Universal Device Driver assigns a unique value. The bulkId assigned is permanent and cannot be altered during this runtime session (*see Bulk Tags section for more information*) (optional).

Tip: If the script defines a bulkId for one tag, it must define the bulkId for every tag. Otherwise, the default value chosen by the Universal Device Driver may conflict with a value previously chosen by the script.

- valid – (Boolean) true if the tag is valid, false if the tag is invalid (required).

onTagsRequest

This function is called when tag values need to be read or written. It is up to the script to determine what action is taken next to execute the read or write and complete the tags. Depending on the protocol being implemented, the tags may be completed right away, additional device I/O may be required, or there may be no way to complete the tags at this time.

Input

onTagsRequest has a single input argument. The argument is a JavaScript Object with the following fields:

- type – (string) the type of operation to perform. The value is Read or Write.
- tags – (Array of Objects) Array of tags being read or written. The tag object has the following fields:
  - address – (string) tag address
  - value – (*) desired value of the tag. This field is only populated when type is Write.
  - dataType – (string) server data type *(see **Data Types** section for valid values)*.
  - readOnly – (Boolean) true for read only; false for read/write tags.

Output

onTagsRequest must return a JavaScript object with the following fields:

- action – (string) This driver's next action. Valid return actions are Receive, Complete, or Fail.
  - Receive – Indicates that the current transaction is not complete and that data is expected imminently from the device.
  - Complete – Indicates that the transaction is complete and no further I/O with the device is needed. If tags were being read, any tags returned with values are completed at this time.
  - Fail – Indicates a serious failure has occurred. All other return fields are ignored, any tags that were requested failed, and the device is in error state.
- data – (Array of numbers) Data to send to the device (optional). If data field is undefined then no data is sent. Values must be between 0 and 255.
- tags – (Array of Objects) Completed tags (optional). If the action is Complete, tags returned in this array will be completed. This field is only required when type is Read. Each Array element has the following fields:
  - address – (string) Tag address
  - value – (*) New value of the tag (optional). If this field is undefined then the tag will be completed with bad quality. Additionally, all tags that are bulked with this tag will be completed with bad quality (*see **Bulk Tags** section for more information*). If value is defined and the quality field is undefined, then the tag will be completed with good quality.
  - quality – (string) Tag quality (optional). Valid quality strings are Good, Bad, or Uncertain.
    - Good – Indicates the quality of the tag value is good. If quality is Good, then the value field must be defined. If the quality field is undefined and the value field is defined, then Good is the default quality.
    - Bad – Indicates the tag value is not useful. A tag value is not required or expected. If the value field is undefined, then Bad is the default quality.

- Uncertain – Indicates the quality of the value of the tag is uncertain. This quality is available to allow the script writer to indicate an issue with the validity, staleness, or out of range state of the value being provided. The value field must also be defined.

onData

This function is called whenever the Universal Device Driver receives data from the device. If there are any uncompleted tags when onData is called, the tags and type (Read or Write) are included along with the data received. If the Universal Device Driver receives data and there are no uncompleted tags, then only the data input field is populated.

***See Also**: Script Utility Functions*

Input

onData has a single input argument. The argument is a JavaScript Object with the following fields:

- data – (Array of numbers) Data that was received by the Universal Device Driver.
- type – (string) Type of operation. The value is Read or Write. This field is undefined if there are no uncompleted tags.
- tags – (Array of Objects) Array of tags being read or written. This field is undefined if there are no uncompleted tags. The tag Object has the following fields:
  - address – (string) Tag address
  - value – (*) Desired value of the tag. This field only exists when type is Write.
  - dataType – (string) server data type *(see Data Types section for valid values)*.
  - readOnly – (Boolean) true for read only; false for read/write tags.

Output

onData must return a JavaScript object with the following fields:

- action – (string) The driver's next action. Valid return actions are Receive, Complete, or Fail.
  - Receive – Indicates that the current transaction is not complete and that more data is expected imminently from the device.
  - Complete – Indicates that the transaction is complete and no further I/O with the device is needed. If tags were being read, then any tags returned with values are completed at this time.
  - Fail – Indicates a serious failure has occurred. All other return fields are ignored, any uncompleted tags fail, and the device is in error state.
- data – (Array of numbers) Data to send to the device (optional). If data field is undefined, no data is sent. Values must be between 0 and 255.
- tags – (Array of Objects) Completed tags (optional). If the action is Complete, tags returned in this array are completed. This field is only required when type is Read. Each Array element has the following fields:
  - address – (string) Tag address.
  - value – (*) New value of the tag (optional). If this field is undefined then the tag will be completed with bad quality. Additionally, all tags that are bulked with this tag will be completed with bad quality (*see Bulk Tags section for more information*). If value is defined and the quality field is undefined, then the tag will be completed with good quality.
  - quality – (string) Tag quality (optional). Valid quality strings are Good, Bad, or Uncertain.
    - Good – Indicates the quality of the tag value is good. If quality is Good, then the value field must be defined. If the quality field is undefined and the value field is defined, then Good is the default quality.

- Bad – Indicates the tag value is not useful. A tag value is not required or expected. If the value field is undefined, then Bad is the default quality.
- Uncertain – Indicates the quality of the value of the tag is uncertain. This quality is available to allow the script writer to indicate an issue with the validity, staleness, or out of range state of the value being provided. The value field must also be defined.

***See Also**: Script Utility Functions, Logging Functions, Bulk Tags*

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)*

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. Global state is saved between function calls so global variables retain their value.
- Be careful of while loops. If while loops are done improperly, they can loop forever and the server can timeout and fail the operation.

***See Also**: Script Utility Functions*

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; Defaults only appear as an input.
- String
- Boolean
- Char
- Byte
- Short
- Word
- Long
- DWord
- Float
- Double
- BCD
- LBCD
- LLong
- QWord

Bulk Tags

To reduce requests to the device, improve throughput from a device, process tags faster, or obtain similar data at the same time, the script can define the bulkId field in the result of the onValidateTag function for each tag. Tags that share the same bulkId are blocked together and provided to the onTagsRequest and onData functions allowing the script to complete all the tags in the block.

If script defines one bulkId, it must define a bulkId for all tags

If the bulkId is not defined in the onValidateTag function, then the Universal Device Driver will assign a unique value to each tag.
Caution: If the script defines the bulkId of one tag, it must define a bulkId for every tag. Otherwise, the value chosen by the Universal Device Driver may conflict with a value previously chosen by the script.

Block Size Performance vs. CPU usage

The onValidateTag function will not limit the number of tags defined with the same bulkId. However, to prevent unnecessary strain on the script engine, the Universal Device Driver will fail Read requests if more than 8000 tags are in the same block.
Caution: The script writer should monitor the performance and CPU usage of the server runtime and script engine processes to ensure that the size of the block is optimal for the use case.

Example with Modbus Protocol

Modbus devices allow requests to specify a certain number of registers starting with a particular address. The script could assign the same bulkId to those tags that could be included in the same request to the device. This reduces the number of transactions required over the wire and improves performance.

Example with MQTT Pub / Sub Protocol

The script could assign the same bulkId to all the tags in the same topic to allow updating them at the same time. This would allow the entire transaction to be processed together, improving performance.

Example with Structure

The tags in a structure often represent the state of the device cycle at a given moment in time. If the tags are updated one at a time, there is no guarantee that the data in one tag was related to the data in another tag. Assigning the same bulkId to those related tags allows the script to issue the necessary requests to the device, save the results, and ultimately update the tags at the same time, ensuring the data is related. For example, if a tag holds the temperature from a temperature sensor and another tag holds the time that was taken, in a non-bulked configuration, the temperature tag could be from a previous read from the time tag. This is dependent on how many tags are in the project and how quickly they are scanned. But if only processing one tag at a time, there is no guarantee of the relationship of the data to other tag data. However, if the tags were assigned the same bulkId, then the script can control the update of the data and ensure that relationship.

Provided Functions

To aid the script writer in writing the profile scripts, the script engine environment (Script Engine SDK) provides the following helper functions and classes:

- Log: allows users to log messages to the server event log
- Cache utility: a cache implementation that stores tag data

Logging Functions

The log() function can be used to log messages to the server event log. 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.

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.

Caching Functions

For unsolicited communications it is necessary to manage a cache so OPC clients can retrieve and update tag data. To help facilitate this there are three caching functions designed to be used in the script: initalizeCache, writeToCache, and readFromCache.

initalizeCache

Initializes the cache. Calling this function allows users to set the maximum cache size.

Input

initalizeCache has one input argument:

- maxSize – (number) the maximum size of the cache (10,000 maximum)

Output

None

writeToCache

Inserts or overwrites a key-value pair in the cache.

Input

writeToCache has two input arguments:

- key – (string) The tag address
- value – (*) The value of the tag. Maximum length of 4096 characters

Output

writeToCache returns a string indicating if the cache was updated successfully. Possible return values are success and error.
Possible causes of an error include the following:

- Any of the arguments passed to the function are undefined or of the wrong type
- The cache size limit has been reached
- The value argument has exceeded its maximum length limit of 4096 characters

readFromCache

Retrieves a key-value pair stored in the cache. If the key-vale pair is not found, the return object's value property has a value of undefined.

Input

readFromCache has one input argument:

- key – (string) The tag address to retrieve data from

Output

readFromCache returns an object that contains the following properties:

- key – (string) The tag address
- value – (*) The last known value of the tag. Undefined if the key is not found in the cache. See below for possible reasons this could occur.

Reasons the value property could be undefined:

- If a key-value pair has not been entered into the cache at the time the readFromCache is called.
- If the cache is full and a key-value pair needs to be added, the cache ages out the first key-value pair updated more than 24 hours prior. Because of this, it is possible that the key being read from the cache is aged out and no longer exists in the cache.

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 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 restart communication and begin reporting "good" quality data again.

Updating a profile can cause linked channels to become invalid. For example, if the onValidateTag function changes and the static tag or dynamic client tag addresses no longer fit the address schema in the new function; those tags that no longer pass validation 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

G

I

L

N

O

P

Q

R

S

T

U

V

W