TOC

Technical Guidance — Configuring Universal Device Driver and Profile Library **3**

(File Access Example) 3

Introduction 3

Key Concepts 3

Installation 4

Understanding the Profile 4

Create a Profile with Kepware Server Configuration 4

Create a Profile with the Configuration API 5

Edit the Script 6

Summary of this script example: 6

Required function: onProfileLoad 6

Required function: onValidateTag 7

Required function: onFileTagsRequest 8

onFileTagsRequestState: onFileTagsRequestStates.initializeCycle 10

onFileTagsRequestState: onFileTagsRequestStates.waitForCycleComplete 10

onFileTagsRequestState: onFileTagsRequestStates.cycleComplete 10

onFileTagsRequestState: onFileTagsRequestStates.waitForCycleCleanup 12

Required function: onFileOperations 12

info.lastOperation: AsyncWatchDir 12

Info.lastOperation: OpenFile 13

info.lastOperation: ReadFile 16

info.lastOperation: CloseFile 17

info.lastOperation: AsyncUnwatchDir 18

Required function: onFileChange 19

Update the Profile 20

Using Server Config 20

Using the Configuration API 20

Create a Channel to Use the Profile 21

Create a Device 21

Create Tags 22

Confirm Example File Profile Script Behavior 22
Congratulations! 23

Technical Guidance —Configuring Universal Device Driver and Profile Library

(File Access Example)

October 2026

Version 1.003

Introduction

This document aims to assist users in the process of setting up the Universal Device Driver to operate on a file or files and understand the concepts of the file access capabilities. This guide uses an example that watches a directory for changes and, if a file is created, the script issues operate requests to open it, write to it, and read from it.

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 velotic.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.).

Key Concepts

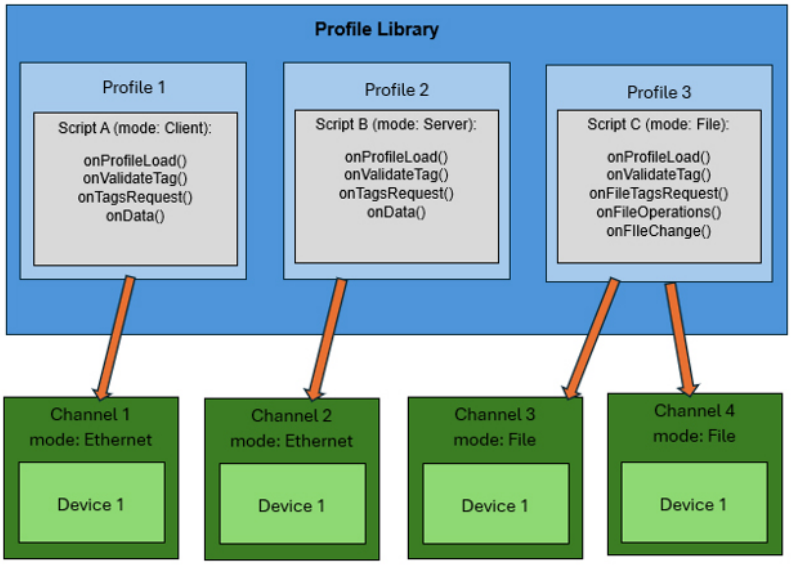
A "profile" exists inside the Profile Library Plug- In and contains a script. Profiles define how the Universal Device driver communicates with a device.

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 Also**: 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.

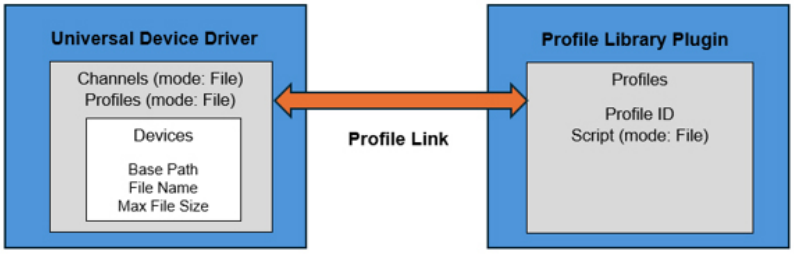
- The Profile Library can contain multiple profiles, each independent of each other.
- Each profile can be linked to multiple channels (1:many).
- Each channel can only be linked to one profile (1:1).
- Each channel maintains its own independent state. Runtime changes (such as creating variables) in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



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.

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 ID is formatted as an auto-generated GUID as 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 knows what script to use to operate on files.

Script: The JavaScript script property is where to define the instructions the driver uses to operate on files. Define the format and logic to build the data to be written to and read from files.

Description: A user-defined description of the profile.

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.
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.

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
```

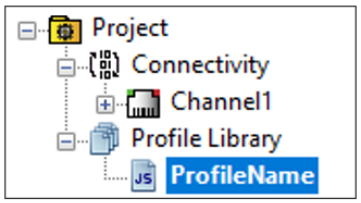
with body:



Tip: This guide uses ModbusProfile as the profile name.

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.
```







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.

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 file operations.
    - **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 use case.
    - **onFileTagsRequest**: Driver prompts the script to process a tag read or tag write. A script can update tags or return file operation requests based on user requirements. For example, a tag may need data from a file or a tag may trigger the script to request to create a file.
    - **onFileOperations**: Interprets the file operation results, issues another file operation request, or updates tag values on success. If the file operation fails, this is indicated in the result provided.
    - **onFileChange**: Interprets the file change results and issues a file operation request or updates tag values.

    Tip: **onFileTagsRequest**, **onFileOperations**, and **onFileChange** can do much more than described in this example. These functions can be used to perform many file operations; i.e. open, close, read, write, create, delete, move, and asynchronously watch for changes.

    Note: Another function must be implemented if the Input String device property is populated.

    For more information, see the Profile Library help for information on the onProfileInputsChange function or the example JavaScripts provided in the install.
3. Build out the script one function at a time, use the following information to edit the script.

Note: the file- profile.js JavaScript provided in the install Examples folder contains similar function definitions. See the readme file installed there for more details on that script.

Summary of this script example:

The script validates tag addresses that are numeric to represent line numbers. The data type of these numerically addressed tags is String to hold each line of text read from the file. The script also allows a Boolean tag with address ResetState, which triggers starting the process again. On the first read tag request, the script starts asynchronously watching the directory found in the Device property **Base Path**. When a new file arrives, the script requests re- opening it. It then requests to write a line of text to the file, which contains several CRLFs. Then it requests to read the file, placing each string, up until a linefeed, into the cache for the line number. Then the script requests to close the file and then to stop watching the dir- ectory. Finally, the script updates the tag(s) from the cache based on the tag address line number and returns the tags to complete the original read request.

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 driver with which it was created, as well as the mode. For more information on the mode, consult the Profile Library Plug- In help. In this example, the mode is File.

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"
- **onFileTagsRequest** causes the read or write operation on the current tag to fail
- **onFileOperations** causes the read or write operation on the current tag to fail
- **onFileChange** causes the read or write operation on the current tag to fail

Global variables are used in this example for readability:

```
/** Global constants */
const PROFILEVERSION = "2.0";
const PROFILEMODE = "File";

// Maximum cache size defaults to 10000 unique key, value pairs.
// Adjust this smaller if do not need to cache that many items.
// Note that the maximum allowed size for each value is 4096.
const MAXCACHESIZE = 10000;
```

Below is the **onProfileLoad** function that indicates the version and mode for **File** and initializes the cache to be used for updating data to be applied to tags when needed:

```
function onProfileLoad() {
log(`onProfileLoad called, initializing cache`, VERBOSE_LOGGING);

initializeCache(MAXCACHESIZE);
return {version: PROFILEVERSION, mode: PROFILEMODE};
}
```

Required function: onValidateTag

The **onValidateTag** script function is to validate the address syntax and data type of a tag. If desired, add logic to this function to modify various tag fields, such as:

- dataType - providing a valid default data type
- address - modifying an address format to enforce consistency among tag addresses
- bulkId - assigning a bulkId to group specific tags together
- arrayColumns - assigning an arrayColumns with an array dataType

For the onValidateTag function in this File example, review the code below:

```
function onValidateTag(info) {
log(`onValidateTag called for address '${info.tag.address}'`, VERBOSE_LOGGING)

let tagAddress = info.tag.address;

// The Reset state tag is a Boolean tag, once the file has been read, write '1' to it to reset the state of the script.
if (tagAddress == "ResetState"){
info.tag.bulkId = TAGTYPERESTSTATE;
info.tag.dataType = "Boolean";
info.tag.valid = true;
return info.tag;
```

```
}

// The tag address for reading a line value is a numeric value representing the one-based line number in the file.
// Supported range is 1 - 255.

try {
let numericAddress = parseInt(tagAddress, 10);
if (numericAddress <= 0 || numericAddress > 0xFF || isNaN(numericAddress)) {
info.tag.valid = false;
return info.tag;
}
info.tag.bulkId = TAGTYPELINE; // All tags blocked together
info.tag.dataType = "String"; // All tags are strings
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);
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.
This function checks the address received from the driver (info.tag.address) and verifies it is either a numeric value between 1 and 255 or the ResetState. If a tag that is not addressed with ResetState has an address that is not within that range or it is not numeric, the function sets the valid field of the tag to false: info.tag.valid = false.
On successful tag address validation, the script also defines the datatype and bulkId fields for each non-array tag. The Universal Device Driver then provides the tags in the tags field of the object passed to the onFileTagsRequest, onFileOperations, and onFileChange functions.

Required function: onFileTagsRequest

The **onFileTagsRequest** script function in this example utilizes the following constants and global variables:

```
// File Actions
const ACTIONCOMPLETE = "Complete";
const ACTIONFAILURE = "Fail";
const ACTIONOPERATE = "Operate";

// File Operations
const OPERATIONOPENFILE = "OpenFile";
const OPERATIONCLOSEFILE = "CloseFile";
const OPERATIONREADFILE = "ReadFile";
const OPERATIONWRITEFILE = "WriteFile";
```

```
const OPERATIONASYNCWATCHDIR = "AsyncWatchDir";
const OPERATIONASYNCUNWATCHDIR = "AsyncUnwatchDir";

// Last Operation Results
const OPERATIONSUCCESSFUL = "Success";

// OpenFile Enumerations
// data[0] - Mode Byte
const OPENMODEREAD = 0;
const OPENMODEWRITE = 1;
const OPENMODEREADWRITE = 2;

// data[1] - Type Byte
const OPENTYPEBINARY = 0;
const OPENTYPETEXT = 1;

// data[2] - Access Byte
const OPENACCESSEXCLUSIVE = 0;
const OPENACCESSDENYWRITE = 1;
const OPENACCESSDENYREAD = 2;
const OPENACCESSDENYNONE = 3;

// data[3] - Create Mode Byte
const OPENCREATEMODEOPENEXISTING = 0;
const OPENCREATEMODECREATETRUNCATE = 1;
const OPENCREATEMODECREATENOTRUNCATE = 2;

// States used in onFileTagsRequest for the cycle executed in this script.
const onFileTagsRequestStates = {
initializeCycle: 0,
waitForCycleComplete: 1,
cycleComplete: 2,
waitForCycleCleanUp: 3
}

const cycleStatuses = {
success: 0,
fail: 1,
closeFileRequired: 2,
asyncUnwatchDirRequired: 3
}

// Text used in WriteFile operation.
const writeText = "Hello\r\nworld.\r\nKeep\r\ncalm\r\nand carry on!";

/** Global Variables */
let onFileTagsRequestState = onFileTagsRequestStates.initializeCycle;
let currentFile = ''
let cycleStatus = cycleStatuses.fail;
```

The **onFileTagsRequest** function returns operate requests or completes the tags, depending on the current value of the global variable onFileTagsRequestState which depends on that state of the cycle of operations.

```
// A cycle consists of the following operations:
// start asynchronously watching the Base Path directory for a created file
// wait for a file to be
// created in the directory
// opened
// written to
// read from, which updates cache with key, value pairs representing file line
// number and string at that file position
// closed
// stop watching the directory
```

onFileTagsRequestState: onFileTagsRequestStates.initializeCycle

```
case onFileTagsRequestStates.initializeCycle:
// Request that the configured Base Path be asynchronously monitored for changes.
cycleStatus = cycleStatuses.fail;
onFileTagsRequestState = onFileTagsRequestStates.waitForCycleComplete;
return {action: ACTIONOPERATE, operations: [OPERATIONASYNCWATCHDIR]};
```

The function updates the onFileTagsRequestState to waitForCycleComplete and returns an action: "Operate" to start asynchronously watching the device Base Path directory.

onFileTagsRequestState: onFileTagsRequestStates.waitForCycleComplete

```
case onFileTagsRequestStates.waitForCycleComplete:
// Waiting for cycle to complete. Fail all tags, except ResetState tag, while waiting.
if (info.tags[0].bulkId != TAGTYPERESETSTATE) {
return {action: ACTIONCOMPLETE};

} else {
// The reset state tag is effectively write only and always good.
// Ignoring write to this tag in this cycle state
info.tags[0].value = false;
info.tags[0].quality = QUALITYGOOD;
return {action: ACTIONCOMPLETE, tags: info.tags};
}
```

While waiting for the cycle of operations to complete from the onFileOperations API function, the quality of each line tag remains "Bad".

onFileTagsRequestState: onFileTagsRequestStates.cycleComplete

```
case onFileTagsRequestStates.cycleComplete: {
// If tag is a line tag
if (info.tags[0].bulkId == TAGTYPELINE) {

```

```
// On success, update value from cache.
if (cycleStatus == cycleStatuses.success) {
for (let i = 0; i < info.tags.length; i++) {
// Read the <one-based line number>, <line text> pair from cache
const result = readFromCache (info.tags[i].address);

if (result.value !== undefined) {
info.tags[i].value = result.value;
log(`Read from cache. Setting value for tag '${info.tags[i].address}' to '${info.tags[i].-
value}' from onFileTagsRequest`, VERBOSE_LOGGING);

} else {
info.tags[i].quality = QUALITYBAD;
log(`Read from cache. No value in cache for tag '${info.tags[i].address}' from onFileOper-
ations. Setting tag quality bad.`, VERBOSE_LOGGING);
}
}
// Complete the tags.
return {action: ACTIONCOMPLETE, tags: info.tags};

// On failure
} else {
return {action: ACTIONCOMPLETE};
}

// If tag is the ResetState tag, check if cycle should start over
} else if (info.tags[0].bulkId == TAGTYPERESETSTATE) {
// If true has been written to the reset state tag, set the script to start the cycle over
if (info.type == "Write" && info.tags[0].value == true) {
log(`Reset cycle from onFileTagsRequest in onFileTagsRequestState ${onFileTag-
sRequestState}.`);
onFileTagsRequestState = onFileTagsRequestStates.initializeCycle;
}

// The reset state tag is effectively write only and always good
info.tags[0].value = false;
info.tags[0].quality = QUALITYGOOD;
return {action: ACTIONCOMPLETE, tags: info.tags};

// Unexpected tag type.
} else {
log(`Error, invalid bulkId '${info.tags[0].bulkId}' on tag to be read`);
return {action: ACTIONFAILURE};
}
}
```

The cycleComplete onFileTagsRequestState indicates that the cycle of operations completed. If any of the operations failed, the cycleStatus variable indicates the failure. If the cycleStatus indicates cycleStatuses.success, the current file has been opened, written to, read from, closed, the directory is no longer being watched for changes, and the cache has been updated from the content in the file. In this

cycleComplete state, the function completes the tags provided in the info structure tags field. If they are the tags with line number addresses and the status of the cycle is success, then the script loops the array of tags, reading the cache for the value at that tag address. If the cache does not contain a value for the tag address yet, the quality of the tag is set to "Bad". If the cache contains a value for the address, the tag value is updated with the value from the cache. If the status of the cycle is not success, then the script returns the action: "Complete" with no tags, which tells the driver they are all bad. If the tag is the ResetState tag and the written value is "True", the onFileTagsRequestState is set to initializeCycle to allow starting the **AsyncWatchDir** operation again. Otherwise, the onFileTagsRequestState remains in the cycleComplete state.

onFileTagsRequestState: onFileTagsRequestStates.waitForCycleCleanup

```
case onFileTagsRequestStates.waitForCycleCleanUp: {
// If an operation failed, file may need to be closed.
if (cycleStatus == cycleStatuses.closeFileRequired) {
return {action: ACTIONOPERATE, operations: [OPERATIONCLOSEFILE], fileOrPath: currentFile};

// To complete the cycle on failure, stop watching the directory.
} else if (cycleStatus == cycleStatuses.asyncUnwatchDirRequired) {
return {action: ACTIONOPERATE, operations: [OPERATIONASYNCUNWATCHDIR]};

} else {
log(`Error, unexpected cycleStatus in onFileTagsRequest '${cycleStatus}'`);
return {action: ACTIONFAILURE};
}
}
```

The script is in this state when one of the following operations failed: OpenFile, WriteFile, ReadFile, or CloseFile. The cycleStatus indicates the action to take for cleaning up. If it is cycleStatuses.closeFileRequired, then the script returns an action: "Operate" to close the file. If the cycleStatus is cycleStatuses.asyncUnwatchDirRequired, then the script returns an action: "Operate" to perform an AsyncUnwatchDir operation.

Required function: onFileOperations

The onFileOperations script function receives the result of a file operation request. In this example, this function checks the lastOperation member of the info object provided to know what operation result it is expecting and what to do next.

info.lastOperation: AsyncWatchDir

```
case OPERATIONASYNCWATCHDIR: {
if (info.lastOperationResult == OPERATIONSUCCESSFUL) {
// AsyncWatchDir succeeded. Monitoring Base Path for changes.
log(`Operation '${info.lastOperation}' on directory '<basepath-
directory>\\${info.lastFileOrPath}' succeeded.`);
return {action: ACTIONCOMPLETE}; // No cache to read from yet; fail tags.

} else {
```

```
// AsyncWatchDir failed. Not monitoring Base Path for changes.
log(`'${info.lastOperation}' failed with reason '${info.lastOperationResult}'`);

// Reset the state to retry the AsyncWatchDir operation
onFileTagsRequestState = onFileTagsRequestStates.initializeCycle;

// Fail all tags except the ResetState tag
if (info.tags[0].bulkId != TAGTYPERESETSTATE) {
return {action: ACTIONCOMPLETE};

} else {
// The reset state tag is effectively write only and always good
info.tags[0].value = false;
info.tags[0].quality = QUALITYGOOD;
return {action: ACTIONCOMPLETE, tags: info.tags};
}
}
}
```

If the result indicates success, the script returns the action: "Complete", which indicates to complete the tags. It does not provide any tags, because the cycle of operations is not yet complete, and the cache does not yet contain newly read data. Therefore, the driver fails the tag. If the result indicates failure, the driver was unable to monitor the specified folder. The script resets the onFileTagsRequestState back to the initializeCycle which again requests the **AsyncWatchDir** operation from the onFileTagsRequest function. The info.lastOperationResult contains a string indicating the reason provided in a log message. For all tags except the ResetState tag, the script returns the action: "Complete" without returning the tags object which indicates to fail the tags. The ResetState tag is returned and always has "Good" quality.

Info.lastOperation: OpenFile

```
case OPERATIONOPENFILE: {
currentFile = info.lastFileOrPath; // Record as current file
if (info.lastOperationResult == OPERATIONSUCCESSFUL) {
// currentFile opened. Write to the file.
log(`Operation '${info.lastOperation}' on file '${currentFile}' succeeded.`);
return { action: ACTIONOPERATE, operations: [OPERATIONWRITEFILE], fileOrPath: currentFile,
data: stringToByteArray (writeText)};

} else {
// Failed to open currentFile. Request to stop watching the directory.
log(`Operation '${info.lastOperation}' on file '${currentFile}' failed with error
'${info.lastOperationResult}'. Failing tags from onFileOperations`);

// On failure to open, fail tags and remember to request to stop watching the folder on the
// next tag request, which is the next opportunity to perform an operation after failure.
onFileTagsRequestState = onFileTagsRequestStates.waitForCycleCleanUp;
cycleStatus = cycleStatuses.asyncUnwatchDirRequired;

// Fail all tags except the ResetState tag
```

```
if (info.tags[0].bulkId != TAGTYPERESETSTATE) {
return {action: ACTIONCOMPLETE};

} else {
// The reset state tag is effectively write only and always good
info.tags[0].value = false;
info.tags[0].quality = QUALITYGOOD;
return {action: ACTIONCOMPLETE, tags: info.tags};
}
}
}
```

If the result indicates success, the script returns the action: "Operate" with the operations: "WriteFile" request. If the file name is provided in the fileOrPath field, it is concatenated with the device **Base Path** property. If the fileOrPath field is undefined, i.e., fileOrPath: "", the driver uses the file name provided in the device **File Name** property. The file must have been previously opened. In this example, the script provides the bytes in the data field. The stringToByteArray helper function defined within the script is used to convert the string defined in the writeText constant to the array of bytes expected in the data field.

```
function stringToByteArray(str) {
let byteArray = [];
for (let i = 0; i < str.length; i++) {
let char = str.charCodeAt(i) & 0xFF;
byteArray.push(char);
}

// return an array of bytes
return byteArray;
}
```

If the result of the operation indicates failure, the driver was unable to open the file. The script sets the onFileTagsRequestState to the waitForCycleCleanup and the cycleStatus to closeFileRequired, which requests the CloseFile operation from the onFileTagsRequest function. For all tags except the ResetState tag, the script returns the action: "Complete" without returning the tags object that indicates to fail the tags. The ResetState tag is returned and always has "Good" quality. The cycle of operations is not yet complete, and the cache does not yet contain newly read data. Therefore, the driver fails the tags.

```
Info.lastOperation: WriteFile
case OPERATIONWRITEFILE: {
currentFile = info.lastFileOrPath; // Record as current file
if (info.lastOperationResult == OPERATIONSUCCESSFUL) {
// currentFile written. Read it back.

// First 4 bytes of info.data contains the number of bytes written from WriteFile.
const bytesWritten = byteArrayToDWord (info.data.slice (0,4));

log(`Operation '${info.lastOperation}' on file '${currentFile}' succeeded. ${bytesWritten}
bytes written.`);
```

```
return {action: ACTIONOPERATE, operations: [OPERATIONREADFILE], fileOrPath: currentFile};

} else {
// Failed to write currentFile.
log(`Operation '${info.lastOperation}' on file '${currentFile}' failed with error
'${info.lastOperationResult}'. Failing tags from onFileOperations`);

// On failure to write, fail tags and remember to request to close the file on the
// next tag request, which is the next opportunity to perform an operation after failure.
onFileTagsRequestState = onFileTagsRequestStates.waitForCycleCleanUp;
cycleStatus = cycleStatuses.closeFileRequired;

// Fail all tags except the ResetState tag
if (info.tags[0].bulkId != TAGTYPERESETSTATE) {
return {action: ACTIONCOMPLETE};

} else {
// The reset state tag is effectively write only and always good
info.tags[0].value = false;
info.tags[0].quality = QUALITYGOOD;
return {action: ACTIONCOMPLETE, tags: info.tags};
}
}
}
```

If the result indicates success, the script returns the action: "Operate" with the operations: "ReadFile" request. If the file name is provided in the fileOrPath field, it is concatenated with the device Base Path property. If the fileOrPath field is undefined, i.e., fileOrPath: "", the driver uses the file name provided in the device File Name property. The file must have been previously opened. In this example, the byteArrayToDWord helper function is used to translate an array of bytes to a 32-bit unsigned integer. It obtains the first four (4) bytes of the info.data field, which contain the number of bytes written to the file. This information is used in the log message posted to the event log.

```
function byteArrayToDWord(byteArray) {
if (byteArray.length == 0 || byteArray.length < 4) {
return 0;
}

const result = (byteArray[3] << 24 | byteArray[2] << 16 | byteArray[1] << 8 | byteArray[0]);
if (result < 0) {
result = 0xFFFFFFFF + result + 1; // Convert negative to positive
}

return result;
}
```

If the result of the operation indicates failure, the driver was unable to write to the file. The script sets the onFileTagsRequestState to the waitForCycleCleanup and the cycleStatus to closeFileRequired, which requests the CloseFile operation from the onFileTagsRequest function. For all tags except the ResetState tag, the script returns the action: "Complete" without returning the tags object which indicates to fail the

tags. The ResetState tag is returned and always has "Good" quality. The cycle of operations is not yet complete, and the cache does not yet contain newly read data. Therefore, the driver fails the tags.

info.lastOperation: ReadFile

```
case OPERATIONREADFILE: {
currentFile = info.lastFileOrPath; // Record as current file
if (info.lastOperationResult == OPERATIONSUCCESSFUL) {
// currentFile read. Update cache and close the file.
log(`Operation '${info.lastOperation}' on file '${currentFile}' succeeded. ${info.data.length} bytes read.`);

// info.data contains the file contents read from ReadFile. Convert this data
// byte array to a string and split into an array of lines (based on linefeed).
// Note: The following is intended for text files but also works on binary files,
// where each line is split on every occurrence of 0x0A.
const dataAsText = byteArrayToString (info.data);
const lines = dataAsText.split('\n');

// Write each line to cache
for (let lineNumber = 1; lineNumber < lines.length + 1; lineNumber++) {
const line = lines[lineNumber - 1]; // Convert to zero-based

log(`Write to cache. Setting key '${lineNumber}' to value '${line}' from onFileOperations`);

// Write the <one-based line number>, <line text> pair to cache
writeToCache(lineNumber.toString (), line);
}

// Although still have to close file and stop watching directory, report
// success since there is data for the tags in the cache now.
cycleStatus = cycleStatuses.success;
return {action: ACTIONOPERATE, operations: [OPERATIONCLOSEFILE], fileOrPath: currentFile};

} else {
log(`Operation '${info.lastOperation}' on file '${currentFile}' failed with error '${info.lastOperationResult}'. Failing tags from onFileOperations`);

// On failure to read, fail tags and remember to request to close the file on the
// next tag request, which is the next opportunity to perform an operation after failure.
onFileTagsRequestState = onFileTagsRequestStates.waitForCycleCleanUp;
cycleStatus = cycleStatuses.closeFileRequired;

// Fail all tags except the ResetState tag
if (info.tags[0].bulkId != TAGTYPERESETSTATE) {
return {action: ACTIONCOMPLETE};

} else {
// The reset state tag is effectively write only and always good
info.tags[0].value = false;
```

```
info.tags[0].quality = QUALITYGOOD;
return {action: ACTIONCOMPLETE, tags: info.tags};
}
}
}
```

If the result indicates success, the info.data contains the information just read from the file. The script loops the data writing each line of data, separated by the linefeed character, to the cache using the line number as the key.

```
function byteArrayToString(data) {
return String.fromCharCode.apply(null, data);
}
```

The byteArrayToString helper function is used to convert the bytes to strings. The script sets the cycleStatus to cycleStatuses.success and returns the action: "Operate" with the operations: "CloseFile" request. If the file name is provided in the fileOrPath field, it is concatenated with the device **Base Path** property. If the fileOrPath field is undefined, i.e., fileOrPath: "", the driver uses the file name provided in the device **File Name** property.
If the info.lastOperationResult of the read file operation indicates failure, the driver was unable to read from the file. The logging message posts the error string to the event log. The script sets the onFileTagsRequestState to the waitForCycleCleanup and the cycleStatus to closeFileRequired, which requests the **CloseFile** operation from the onFileTagsRequest function. For all tags except the ResetState tag, the script returns the action: "Complete" without returning the tags object which indicates to fail the tags. The ResetState tag is returned and always has "Good" quality. The cycle of operations is not yet complete, and the cache does not yet contain newly read data. Therefore, the driver fails the tags.

info.lastOperation: CloseFile

```
case OPERATIONCLOSEFILE: {
currentFile = info.lastFileOrPath; // Record as current file
if (info.lastOperationResult == OPERATIONSUCCESSFUL) {
// currentFile closed
log(`Operation '${info.lastOperation}' on file '${currentFile}' succeeded.`);
return {action: ACTIONOPERATE, operations: [OPERATIONASYNCUNWATCHDIR]};

} else {
// Failed to close currentFile; ignored.
log(`Operation '${info.lastOperation}' on file '${currentFile}' failed with error
'${info.lastOperationResult}'. Failing tags from onFileOperations`);

// On failure to close, fail tags and remember to request to stop watching the folder on
// the next tag request, which is the next opportunity to perform an operation after failure.
// If file failed to close and the script does not try again to close it on the next
// opportunity to perform an operation, the file is closed when the server runtime shuts down.
onFileTagsRequestState = onFileTagsRequestStates.waitForCycleCleanUp;
cycleStatus = cycleStatuses.asyncUnwatchDirRequired;

// Fail all tags except the ResetState tag
```

```
if (info.tags[0].bulkId != TAGTYPERESETSTATE) {
return {action: ACTIONCOMPLETE};

} else {
// The reset state tag is effectively write only and always good
info.tags[0].value = false;
info.tags[0].quality = QUALITYGOOD;
return {action: ACTIONCOMPLETE, tags: info.tags};
}
}
}
```

If the result indicates success, the script returns the action: "Operate" with the operations: "AsyncUnwatchDir" request.
If the info.lastOperationResult of the close file operation indicates failure, the driver was unable to close the file. The logging message posts the error string to the event log. The script sets the onFileTagsRequestState to the waitForCycleCleanup and the cycleStatus to asyncUnwatchDirRequired, which requests the **AsyncUnwatchDir** operation from the onFileTagsRequest function. For all tags except the ResetState tag, the script returns the action: "Complete" without returning the tag object, which indicates to fail the tags. The ResetState tag is returned and always has "Good" quality. The cycle of operations is not yet complete and the cache does not yet contain newly read data. Therefore, the driver fails the tags.

info.lastOperation: AsyncUnwatchDir

```
case OPERATIONASYNCUNWATCHDIR: {
if (info.lastOperationResult == OPERATIONSUCCESSFUL) {
// AsyncUnwatchDir succeeded. No longer monitoring Base Path for changes.
log(`Operation '${info.lastOperation}' succeeded.`);

} else {
// AsyncUnwatchDir failed; log and ignore
log(`'${info.lastOperation}' failed with reason '${info.lastOperationResult}'`);
}

// Regardless of the outcome for AsyncUnwatchDir, cycle is complete.
onFileTagsRequestState = onFileTagsRequestStates.cycleComplete;

// If all operations succeeded
if (cycleStatus == cycleStatuses.success) {
// Update the line tags
if (info.tags[0].bulkId == TAGTYPELINE) {
// Read from cache, update tags, and complete
for(let i=0; i < info.tags.length; i++) {
// Read the <one-based line number>, <line text> pair from cache
const result = readFromCache(info.tags[i].address);

if (result.value !== undefined) {
info.tags[i].value = result.value;
log(`Read from cache. Setting value for tag '${info.tags[i].address}' to '${info.tags[i].-
value}' from onFileOperations`);
```

```
} else {
info.tags[i].quality = QUALITYBAD;
log(`Read from cache. No value in cache for tag '${info.tags[i].address}' from onFileOper-
ations. Setting tag quality bad.`);
}
}
} else if (info.tags[0].bulkId == TAGTYPERESETSTATE) {
// The reset state tag is effectively write only and always good
info.tags[0].value = false;
info.tags[0].quality = QUALITYGOOD;

} else {
info.tags[0].quality = QUALITYBAD;
}

// One of the operations failed, fail all tags except the ResetState
} else {
if (info.tags[0].bulkId != TAGTYPERESETSTATE) {
return {action: ACTIONCOMPLETE};

} else {
// The reset state tag is effectively write only and always good
info.tags[0].value = false;
info.tags[0].quality = QUALITYGOOD;
}
}

return {action: ACTIONCOMPLETE, tags: info.tags};
}
```

A message is logged to alert the user of the result of the **AsyncUnwatchDir** file operation request. The onFileTagsRequestState is changed to cycleComplete. If all operations in the cycle have succeeded, then cycleStatus indicates that success. The line tags are updated from the cache, and the script returns the action: "Complete" with tags: info.tags finally returning the tags. If any operation in the cycle failed, then fail all tags except the ResetState tag.

Required function: onFileChange

The onFileChange function parses the info structure for the detected change reason in the info.change member and the file or directory that changed in the info.fileOrPath member. This example logs a message indicating the file or path and detected change reason.

```
function onFileChange(info) {
// Handle asynchronous change to info.fileOrPath
log(`onFileChange called. Detected file change in onFileTagsRequestState ${onFileTag-
sRequestState}: FileOrPath '${info.fileOrPath}' was '${info.change}'.`);

if (onFileTagsRequestState == onFileTagsRequestStates.waitForCycleComplete &&
info.change == "Created") {
```

```
// info.fileOrPath created, next step of the cycle in this example script is to open the file
log(`Requesting to open '${info.fileOrPath}' from onFileChange.`);
return {action: ACTIONOPERATE, operations: [OPERATIONOPENFILE], fileOrPath: info.fileOrPath,
data: [OPENMODEREADWRITE, OPENTYPEBINARY, OPENACCESSEXCLUSIVE,
OPENCREATEMODECREATENOTRUNCATE]};

} else {
// Ignoring all other file or path changes
log(`Ignoring file or path change from onFileChange.`, VERBOSE_LOGGING);
return {action: ACTIONCOMPLETE}; // Won't fail the tags
}
}
```

The function ignores the change reason if the onFileTagsRequestState is not waitForCycleComplete and the change is not “Created”. If a file is created, the script issues the OpenFile operation request on that file with the four bytes of the info.data field set to the open flags for read and write access, binary mode, denying read access to others, and to create a new file if no file exists, otherwise open existing.

Update the Profile

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

Using Server Config

1. Open the property editor for the new profile created in this exercise.
2. Load the profile in the **Script** property field by copying the content of entire JavaScript file and paste it into the field in the property editor OR clicking the ellipses (..) at the right of the Script property field 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 a text editor to remove all line endings or convert them to Linux (\n character) line endings.

3. Click **OK**.

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 saved.

   Note: The 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.

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. Select **File** for the mode.
6. Leave all other values at the defaults, click **Next**.
7. Click **Finish**.

Create a Device

Configure a device under the new channel. This guide leaves most settings at the defaults, but when the parent channel is in File Mode, the device must have a valid Base Path configured.

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 Base Path, specify the directory where files are located. The directory must exist for the file operations in the script to succeed.
6. For the File Name; if the script does not provide the file name(s) for file operations, enter the file name. If the script provides the file name(s), this property can be left blank. If both the property and script provide a file name, the driver uses the file name provided by the script. In this example, the script provides the file name.
7. Accept the default for **Max File Size**.
8. Click **Next**, then **Finish**.

9. Verify that the following message is in the Event Log:

```
Script Engine service starting.
Profile link established. | Channel = 'UDDFileModeChannel', profile = 'ProfileName',
state machine version = '2.0', mode='File'.
```

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.

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 1 for this guide. This represents the first line in the file.
5. For the **Data Type**, select **String** from the drop-down and click **OK**.

Confirm Example File Profile Script Behavior

1. Launch **OPC QuickClient** from the configuration.
2. Verify the new tag shows bad quality data. The script does not provide a value or quality of the tag until a file has been opened, written to, and read. If the string entered in the Device Base Path property exists, the file operation request issued in the onFileTagsRequest to watch that directory should succeed. This file_profile.js script is written to log information for the purposes of this example. The event log should indicate that the Operation AsyncWatchDir on the directory succeeded.

   Note: The actual Base Path used is not provided back to the script.

3. The driver is now watching the directory for file changes. When files in the directory change (i.e. "Created", "Deleted", "RenamedFrom", "RenamedTo", or "Modified"), the driver notifies the script by calling the required onFileChange API function. The script, which is written to log information for purposes of this example, logs the detected change.

   Tip: Users can change the script to react to the changes detected. This example script reacts to a created file and issues a file operation request to open it.

4. Once the onFileOperations API function is called with status of Success for the OpenFile operation request, this example script issues a WriteFile operation request to write the "Hello\r\nworld.\r\nKeep\r\ncalm\r\nand carry on!" text to the file. The script uses the stringToByteArray helper function to convert the string into bytes for the data field.
5. Once the onFileOperations API function is called with status of Success for the WriteFile operation request, this example script issues a ReadFile operation request.
6. Once the onFileOperations API function is called with the status of Success for the ReadFile operation request, this example script uses the byteArrayToString helper function to convert the data bytes to string

and writes each line that was read from the file to the cache using the line number of the file as key in the cache. This example script issues a CloseFile operation request.

7. Once the onFileOperations API function is called with the status of Success for the CloseFile operation request, this example script issues a file operation request to stop watching the Base Path. At the same time, the script reads the data from the cache to update the info.tags and completes the tags.

Congratulations!

You completed the process of setting up the Universal Device Driver to operate on a file or files. Hopefully this example provides a foundation for future custom operations and scripts to provide solutions in your environment.

Proprietary NoticeThis publication is provided by Velotic™and/or its affiliates for informational purposes only and does not constitute legal, security or financial advice. The contents of the publication are based on information, data and sources made available by you to Velotic™and while every effort has been made to ensure the accuracy of the report, Velotic™does not guarantee the accuracy, completeness, or reliability of the information contained herein. No representations or warranties, express or implied, are made as to the accuracy, completeness, or reliability of the information contained herein and Velotic™ does not accept any responsibility for any loss or damage arising from the use of, or reliance on, the information presented in this report. You acknowledge that reliance upon the contents of this report is at your sole risk.No part of this publication may be reproduced in any form, or stored in a database or retrieval system, or transmitted or distributed in any form by any means, electronic, mechanical photocopying, recording or otherwise, without the prior written permission of Velotic™. Information contained herein is subject to change without notice.©2026 Velotic™and/or its affiliates. All rights reserved. Trademark NoticesMicrosoft®is a registered trademark of Microsoft Corporation, in the United States and/or other countries.All other trademarks are the property of their respective owners.