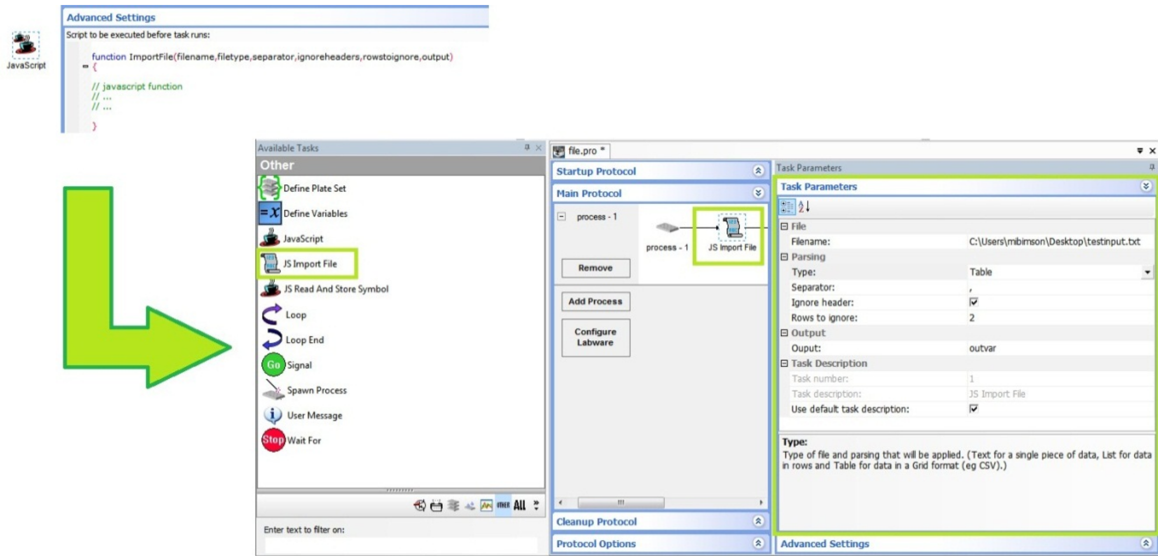
Agilent Technologies

# Using the JavaScript Wrapper and the Global Script features in VWorks Automation Software, Version 11.4 and Later

## JavaScript wrapper

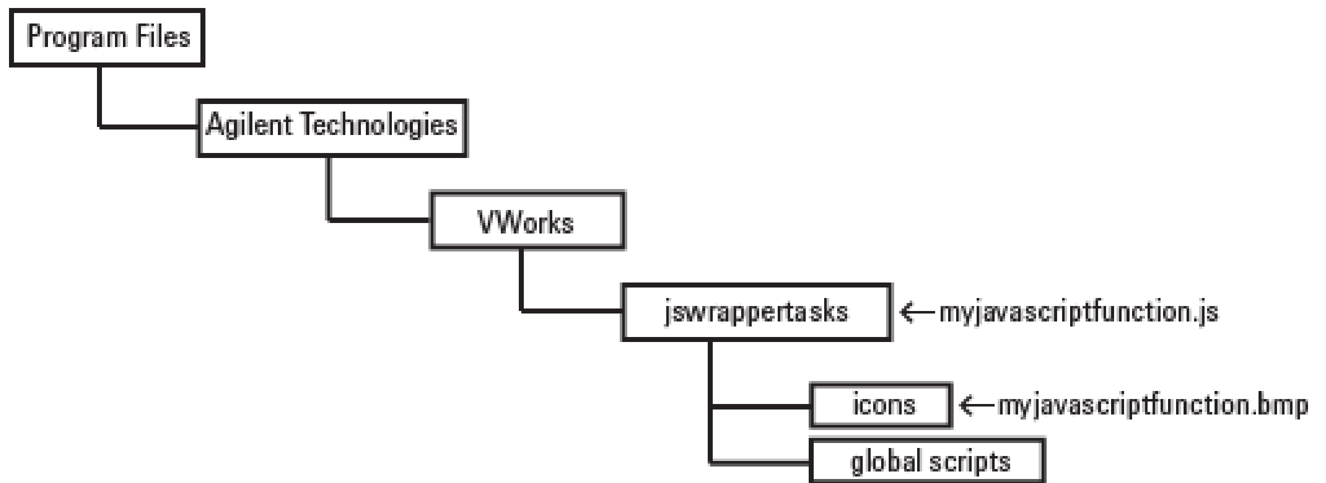
### Introduction

The VWorks JavaScript wrapper feature, available in VWorks 11.4 or later, allows you to create new protocol tasks. The feature enables you to display new task icons and task parameters in the VWorks software user interface.




### File locations

The JavaScript file containing the function to be wrapped has to be placed in a specific location relative to the VWorks directory (default, C:\Program Files\Agilent Technologies\VWorks\jswrappertasks). The file name should be the same as the JavaScript function. An icon file has to be placed in a separate directory (within jswrappertasks directory) with the same name as the JavaScript function (see below).

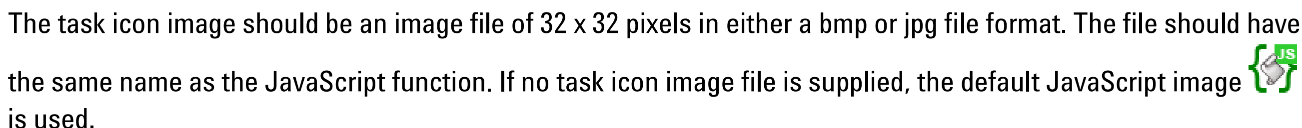



For example, for a function named **myjavascriptfunction,** create a file named **myjavascriptfunction.js** in the jswrappertasks directory and a task icon file **myjavascriptfunction.bmp** in the icon directory.

The VWorks software uses these files to create the JavaScript wrapper tasks during startup. Therefore, any changes to the XML command block within the JavaScript file must take place before the VWorks software is started.

Agilent Technologies

## Icon file

The task icon image should be an image file of 32 x 32 pixels in either a bmp or jpg file format. The file should have the same name as the JavaScript function. If no task icon image file is supplied, the default JavaScript image  is used.

## JavaScript file

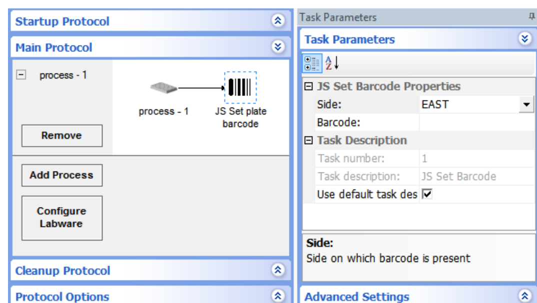
The JavaScript file should have the file extension .js. It contains two parts: the XML command information, which describes the task function, task parameters, GUI, and the JavaScript function which will execute. Below is an example of a file that creates a task to set a labware’s barcode.

```
///<?xml version='1.0' encoding='ASCII' ?>
///<Velocity11 file='MetaData' md5sum='00000000000000000000000000000000' version='1.0' >
///     <Command Compiler='0' Editor='2' Name='JS Set Barcode' Description='JS Set plate barcode' >
///             <Parameters >
///                     <Parameter Name='Side' Description='Side on which barcode is present'
Scriptable='0' Type='2' >
///                             <Ranges >
///                                     <Range Value='NORTH' />
///                                     <Range Value='SOUTH' />
///                                     <Range Value='EAST' />
///                                     <Range Value='WEST' />
///                             </Ranges>
///                     </Parameter>
///                     <Parameter Name='Barcode' Description='Value of barcode to be set'
Scriptable='1' Type='1' />
///             </Parameters>
///     </Command>
///</Velocity11>

function jsAddin_setBarcode(jsAddin_setBarcode_parameter1, jsAddin_setBarcode_parameter2)
{
// convert jsAddin_setBarcode_parameter1 text to variable name
jsAddin_setBarcode_side = eval(jsAddin_setBarcode_parameter1)

// set barcode
plate.setBarcode(jsAddin_setBarcode_side, jsAddin_setBarcode_parameter2)
}
```

This produces a task that looks like the following.




In the above the function, jsAddin_setBarcode() requires two arguments (jsaddin_setBarcode_parameter1 and jsaddin_setBarcode_parameter2).

It is important to use long unique names for all functions and variables as once the JavaScript wrapper is executed, they are global and could otherwise conflict with functions and variables defined in the protocol. Using the naming structure, such as “jsAddin_functionname()” and “jsAddin_functionname_variablename” should lead to unique names.

The XML block describes the task to be created and how arguments are entered. The basic structure is shown below.

```
///<?xml version='1.0' encoding='ASCII' ?>    XML declaration
///<Velocity11>                               Velocity 11 root element
///         <Command>                         Command element describing the task
///                 <Parameters >             Parameters element describing how arguments are displayed
and entered
///                         <Parameter />     Escaped Parameter element describing how 1st argument is
entered in task parameters GUI
///                         <Parameter />     Escaped Parameter element describing how 2nd argument is
entered in task parameters GUI
///                 </Parameters>             Close Parameters element
///         </Command>                        Close Command element
///</Velocity11>                              Close Velocity11 element
```

The XML block uses /// at the start of each line to distinguish it from the JavaScript.

The XML elements are fully described in the VWorks Plugin Developers Guide (G5415-90065) in the section on common elements and attributes. However, a summary of common attributes is shown below.

| Element | Attribute | Value | Description |
|---|---|---|---|
| Velocity11 | File | MetaData | Specifies XML structure (in this case use MetaData). |
| | md5sum | 32 digit hex nnumber | 128- bit hash value that can be used to verify the integrity of an XML block. Use 32 zeros until md5sum calculated. |
| | version | 1.0 | Currently always 1.0. |
| Command | Name | Text | Name of task (as seen in available task list). |
| | Description | Text | Description of task (as seen under the task in a protocol). |
| | Compiler | Bitmask | Compiler options.<br>0 indicates no compiler action |
| | Editor | Bitmask | Protocol editor options:<br>0 = disregard (default)<br>1 = hide if not available<br>2 = Main protocol editor<br>4 = Sub-process editor<br>8 = Startup/cleanup process editor<br>16 = All editors |
| Parameters | | | Contains one or more Parameter elements. |
| Parameter | Name | Text | Name of field (parameter). |
| | Description | Text | Description of field (shown in task parameters when field is selected) |
| | Category | Text | A name used to group two or more Parameter elements. |
| | Hide_if | Logical test | When the value of the conditional expression in the Hide_if attribute is true, the parameter field is hidden or made read-only.<br>Example<br>`Hide_if='Variable(Type)!=Const(Table)'`<br>The parameter will be hidden if the variable Type (the name of a parameter in the same category) is not equal to the constant (in this case the word Table). |

| Parameter | Scriptable | Number | Indicates if the parameter field can accept JavaScript in the editing dialog.<br>0 = not scriptable |
|---|---|---|---|
| | Style | Number | How parameter fields are displayed in the task window.<br>0 = displayed as read/write (default)<br>1 = displayed as read only<br>2 = displayed as read only and hidden when hide disabled tasks selected in VWorks options. |
| | Type | Number | Describes the type of field displayed<br>0 = Provides a Boolean check box.<br>1 = Allows the user to specify a character string.<br>2 = Provides a drop- down list box.<br>3 = Provides a drop- down combo box.<br>4 = Allows the user to specify a device location.<br>5 = Allows the user to specify a labware or a fixed location.<br>6 = Allows the user to specify both a location and the labware to use.<br>7 = Opens the Well Selection dialog box.<br>8 = Allows the user to specify an integer.<br>9 = Allows the user to specify a file path.<br>10 = Provides a labware drop- down list box.<br>11 = Provides a liquid- class drop- down list box.<br>12 = Allows the user to specify a decimal fraction.<br>13 = Allows the user to specify a file path, where the value can be empty.<br>14 = Allows the user to enter a password and displays a series of asterisks to hide the password string.<br>15 = Allows the user to specify an IP address.<br>16 = Allows the user to select a directory.<br>17 = Allows the user to enter a time in the format hh:mm:ss.<br>18 = Refers to an object in the JavaScript scripting context.<br>19 = Allows the user to enter a date. The format depends on the region and language settings.<br>20 = Allows the user to enter character strings that can wrap onto multiple lines.<br>21 = Opens the Pipette Technique Editor.<br>22 = Opens the Head Mode Selector dialog box.<br>23 = Describes the tip positions of a tip box.<br>24 = Opens the Field Composer dialog box.<br>25 = Displays the available hit pick format files. For example, when the user clicks the down arrow in the Format file field of the Hit pick replicate task, the list that is displayed is of this type.<br>26 = Deprecated. Used to show the available analog input names in the device file where the plugin resides.<br>27 = Deprecated. Used to show the available digital input names in the device file where the plugin resides.<br>28 = Deprecated. Used to show the available digital output names in the device file where the plugin resides.<br>29 = Converts a parameter of this type to, and accesses it as, a JavaScript array object.<br>30 = Allows the user to specify a duration in the format n Days hh:mm:ss.<br>31 = Displays a multi- line text box.<br>32 = Opens the color palette that enables the user to change the colors of various dialog box components. |

| Parameter | Value | Dependent on Type | Default value for parameter. |
|---|---|---|---|
| Ranges | | | Contains one or more Range elements. |
| Range | Value | Text | When the Parameter's Type attribute is 2 or 3 then each Range element and Value attribute describe a menu item that can be selected.<br>When the Parameter's Type attribute is 8 or 12 then two Range elements with Value attributes containing integers describe the maximum and minimum possible values. |

## Designing the task parameters GUI

### Using different field types

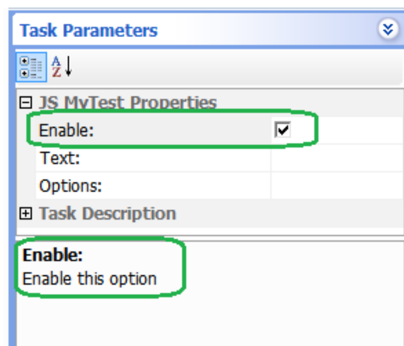
Below are examples of the three commonest types of field for input.

#### *Check box*

The check box is displayed for a parameter when ParameterType ='0'. When selected it has a value of 1, and when not selected has a value of 0.

For example, the following xml parameter element produces the display below.

```
/// <Parameter Name='Enable' Description='Enable this option' Type='0' Value='1'/>
```



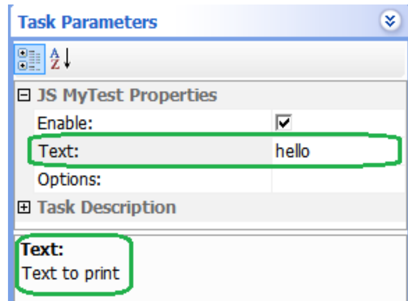

The Value attribute specifies the default starting condition. In the above case it is checked.

### *Edit field*

The edit field is displayed for a parameter when Type ='1'. It passes a character string to the functions argument.

For example the following xml parameter element produces the display below.

```
/// <Parameter Name='Text' Description='Text to print' Type='1' />
```

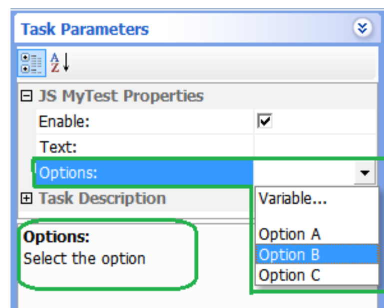



### *Drop List*

The drop list field is displayed for a parameter when Type ='2'. It passes a character string of the selected item to the functions argument. The items in the drop list are defined by the Ranges and Range elements.

For example the following xml parameter element produces the display below.

```
///   <Parameter Name='Options' Description=Select the option' Type='2' >
///              <Ranges>
///                      <Range Value='Option A' />
///                      <Range Value='Option B' />
///                      <Range Value='Option C' />
///              </Ranges>
///   </Parameter>
```



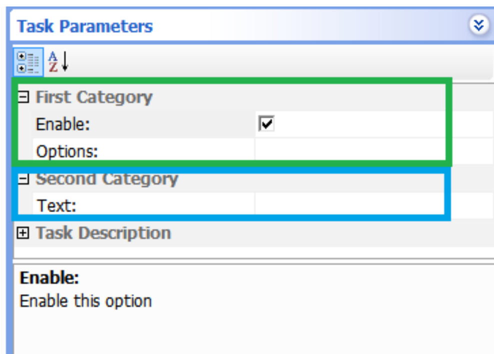

Above shows the options available.

## Grouping fields in the task parameter GUI

Fields in the task parameters GUI can be grouped together under different headings using the Category attribute.

The Category attribute will place the fields in the order the Categories are first defined and the order each parameter is defined. If no Category is defined for a parameter it is placed in a Category with the name of the task.

```
///     <Parameter Name='Enable' Description='Enable this option' Type='0' Value='0' Category='First
Category' />
///     <Parameter Name='Text' Description='Text to print' Type='1' Category='Second Category' />
///     <Parameter Name='Options' Description='Select the option' Type='2' Category='First Category' >
///             <Ranges>
///                     <Range Value='Option A' />
///                     <Range Value='Option B' />
///                     <Range Value='Option C' />
///             </Ranges>
///     </Parameter>
```




## Enabling and disabling fields based on user selections

The Hide_if attribute allows you to enable or disable a field based on the value of another field. The syntax is

```
Hide_if='[Item1] [logical test] [Item2]',
```

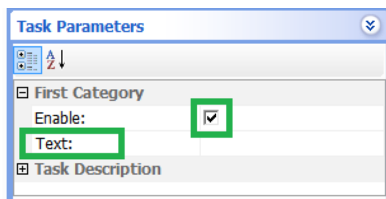
where the field is disabled if the outcome of the logical test is true.

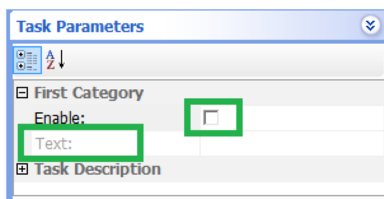
For example below, the field is disabled if the parameter named Input (this is a variable) is not equal (!=) to the character string "Option A " (this is a constant). The logical tests are similar to JavaScript (==,!=,>,<...).

```
Hide_if='Variable(Input)!=Const(Option A)'
```

The Hide_if attribute only allows variables to be used from within the same Category. If the Enable check box is not selected the Text field is disabled (see below).

```
///     <Parameter Name='Enable' Description='Enable this option' Type='0' Value='0' Category='First
Category' />
///     <Parameter Name='Text' Description='Text to print' Type='1' Category='First Category'
Hide_if='Variable(Enable)==Const(0)'/>
```





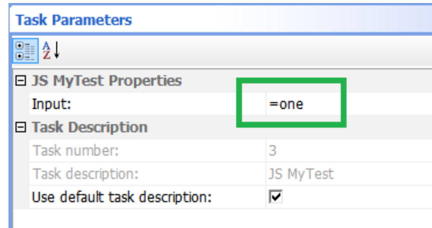
## Passing JavaScript variables and arrays to and from the function

It is possible to pass data from JavaScript variables/arrays within the protocol to the JavaScript function of a JavaScript wrapper task. A mechanism also exists to return data from the JavaScript wrapper task.

### Fields are scriptable

By default fields are scriptable. The field accepts either the' = javascript variable' or display the option 'Variable...'.

```
///     <Parameter Name='Input' Type='1' />
```



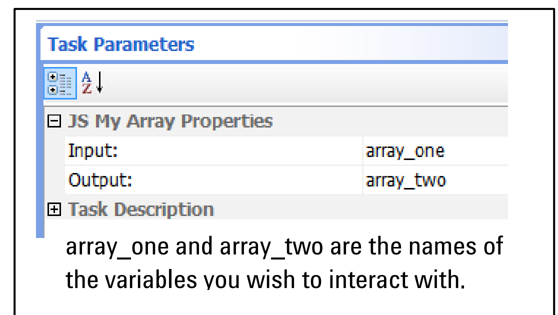

In the above example, the contents of the protocol variable *one* is passed to the JavaScript wrapper parameter.

### Using the global scope

Protocol variables are accessible from within the JavaScript wrapper function. This allows you to act on them directly. To make JavaScript wrapper tasks generic (where the names of the protocol variables are not known when writing the JavaScript wrapper task), you can pass the protocol variable or array names to the JavaScript wrapper in an edit field during protocol writing.

```
///<?xml version='1.0' encoding='ASCII' ?>
///<Velocity11 file='MetaData' md5sum='00000000000000000000000000000000' version='1.0' >
///                <Command Compiler='0' Name='JS My Array' Description='JS My Array' Editor='-1'
NextTaskToExecute='1' RequiresRefresh='0'
TaskRequiresLocation='0' VisibleAvailability='1' >
///                 <Parameters >
///                        <Parameter Name='Input' Description='Name of array to send to function'
Type='1' />
///                        <Parameter Name='Output' Description='Name of array returned from function'
Type='1' />
///                 </Parameters>
///                </Command>
///</Velocity11>

function myarray(jsAddin_myarray_input,jsAddin_myarray_output)
{
 // Assign input data to temp
 eval("jsAddin_myarray_temp = "+jsAddin_myarray_input)
 // Transfer temp to output
 eval(jsAddin_myarray_output+" = jsAddin_myarray_temp")
}
```




array_one and array_two are the names of the variables you wish to interact with.

The character string entered for the variable or array name is converted into an actual variable or array name in JavaScript using the function eval().

The function eval() processes a string as JavaScript.

This mechanism allows data to be passed to and from the JavaScript wrapper task without reserving specific variable/array names.

## Example: File import and parsing task

The following example creates a file import and parsing task. It has the following key features.

- The import file can be selected.
- The file type can be specified (text/list/table).
- For the file type table the column separator can be specified (,/;/:/TAB).
- For the file types list and table the header lines can be ignored.
- The output can be specified (name of variable or array).

With an appropriate icon file, the task parameters appears as follows in the user interface.



The JavaScript wrapper file to generate this task is shown below.

```
///<?xml version='1.0' encoding='ASCII' ?>
///<Velocity11 file='MetaData' md5sum='00000000000000000000000000000000' version='1.0' >
///                <Command Compiler='0' Description='JS Import File' Editor='-1' Name='JS Import
File' NextTaskToExecute='1' RequiresRefresh='0' TaskRequiresLocation='0' VisibleAvailability='1'
>
///                <Parameters >
///                        <Parameter Name='Filename' Description='Filename' Category='File'
Scriptable='1' Type='9' />
///                        <Parameter Name='Type' Description='Type of file and parsing that will be
applied. (Text for a single piece of data, List for data in
rows and Table for data in a Grid format (eg CSV).)' Category='Parsing' Type='2' Scriptable='0'
Value='Text'>
///                                <Ranges>
///                                        <Range Value='Text' />
///                                        <Range Value='List' />
///                                        <Range Value='Table' />
///                                </Ranges>
///                        </Parameter>
///                        <Parameter Name='Separator' Description='Separator for Grid file type '
Category='Parsing' Type='2' Scriptable='0' Value=','
Hide_if='Variable(Type)!=Const(Table)' >
///                                <Ranges>
///                                        <Range Value=',' />
///                                        <Range Value='TAB' />
///                                        <Range Value=';' />
///                                        <Range Value=':' />
///                                </Ranges>
///                        </Parameter>
///                        <Parameter Name='Ignore header' Description='Ignore header rows in
list and grid files.' Category='Parsing' Type='0' Scriptable='1'
Hide_if='Variable(Type)==Const(Text)' />
///                        <Parameter Name='Rows to ignore' Description='Number of rows to ignore in
list and grid files.' Category='Parsing' Type='1' Scriptable='1'
Hide_if='Variable(Type)==Const(Text)' />
///                        <Parameter Name='Ouput' Description='Javascript variable or array into which
data is passed. (For Text this results in a variable
(eg x), for List this results in an array (eg x[row]) and for Table this results in a 2D array (eg
x[row][column]).)' Category='Output' Type='1' />
///                </Parameters>
///                </Command>
///</Velocity11>

Function
jsAddin_ImportFile(jsAddin_ImportFile_filename,jsAddin_ImportFile_filetype,jsAddin_ImportFile_separator,
jsAddin_ImportFile_ignoreheaders,jsAddin_ImportFile_rowstoignore,jsAddin_ImportFile_output)
{
// create file object
jsAddin_ImportFile_fo = new File()

// open file
jsAddin_ImportFile_fo.Open(jsAddin_ImportFile_filename)

// read file into variable contents
jsAddin_ImportFile_contents = jsAddin_ImportFile_fo.Read()

// close file
jsAddin_ImportFile_fo.Close()

// Parse contents depending file type/parsing method
switch(jsAddin_ImportFile_filetype)
{

// Text - assumes one value and passes directly to variable jsAddin_ImportFile_temp case "Text":
jsAddin_ImportFile_temp = jsAddin_ImportFile_contents
break;
// List - assumes a list with a single piece of data on each row
case "List":
jsAddin_ImportFile_temp = jsAddin_ImportFile_contents.split("\n")
break;
// Table - assumes a tabular data structure in row and columns
case "Table":
jsAddin_ImportFile_temp = new Array()
jsAddin_ImportFile_lines = jsAddin_ImportFile_contents.split("\n")
if(jsAddin_ImportFile_ignoreheaders=="1")
{
```

```
jsAddin_ImportFile_start = jsAddin_ImportFile_rowstoignore
}
else
{
jsAddin_ImportFile_start = 0
}
jsAddin_ImportFile_ii = 0
for(jsAddin_ImportFile_i=jsAddin_ImportFile_start;jsAddin_ImportFile_i<jsAddin_ImportFile_lines.length;
jsAddin_ImportFile_i++)
{
jsAddin_ImportFile_temp[jsAddin_ImportFile_ii]=jsAddin_ImportFile_lines[jsAddin_ImportFile_i].split(jsAdd
in_ImportFile_separator)
jsAddin_ImportFile_ii++
}
break;
}

// assign jsAddin_ImportFile_temp to output variable
eval(jsAddin_ImportFile_output +" = jsAddin_ImportFile_temp")
}
```

# Global script file

## Introduction

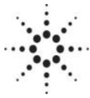
The global script file jsaddin_global_vars.js located in the directory globalScripts allows us to declare functions and variables that will be accessible to a JavaScript wrapper task or any protocol that has a JavaScript wrapper task.

## Use

You can add functions and variables to the file before the VWorks software is launched. This includes using the open() command to process other JavaScript files.

The functions and variables defined in the global script file are available to any protocol that contains a JavaScript wrapper. Unlike the functions and variables described in a JavaScript wrapper, they are available before the JavaScript wrapper task is actually executed.

To minimize conflict of names it is prudent to use long unique names for variables and functions such as “jsAddin_functionname()” and “jsAddin_variablename”.



Agilent Technologies

## Enabling Global Scripts

The Global script file is ONLY processed when the protocol contains a JavaScript wrapper task.

If you only want to use the Global script file and not a specific JavaScript wrapper task it is possible to prepare a JavaScript wrapper task whose sole function is to enable the Global script file. Create a JavaScript wrapper file called jsAddin_enableGlobalScripts.js containing the following.

```
///<?xml version='1.0' encoding='ASCII' ?>
///<Velocity11 file='MetaData' md5sum='00000000000000000000000000000000' version='1.0' >
///     <Command Compiler='0' Editor='16' Name='JS Enable Global Scripts' Description='JS Enable Global Scripts File (jsaddin_global_var.js)' >
///             <Parameters />
///     </Command>
///</Velocity11>

function jsAddin_enableGlobalScripts()
{
print("Global Script File Enabled")
}
```

Refer to the section on JavaScript wrapper tasks for more information.


Copyright© Agilent Technologies, Incorporated 2013 All rights reserved