TwinCAT ADS Made Easy

TAME 2.2

JavaScript Library

API Reference

TAME 2.2: API Reference

Last changed: 03.05.2012

Chapters

1. Introduction

TAME is JavaScript library created for an easy and comfortable access to the TwinCAT ADS WebService. The name is an acronym for „TwinCAT ADS Made Easy“ and stands also for „taming“ the complexity of ADS and AJAX requests. Originally a „wast product“ from the programming of a browser based visualisation for my home, it has become a feature rich piece of software and I hope it will be useful for others. It allows to exchange data with a TwinCAT PLC without any knowledge of ADS. The communication is based on AJAX requests, the browser connects to the webserver running on the PLC device. If you want to send or read data through a firewall you need to forward only one port (80 by default) to connect to the WebService.

There are methods for read and write access to single variables, variable blocks, arrays and structures in the TwinCat PLC. To reduce overhead and keep the communication efficent, access is only possible to allocated addresses (M/MX,I/IX,Q/QX), there is no access by handles.

Supported data types are BOOL, BYTE, WORD, DWORD, USINT, SINT, UINT, INT, UDINT, DINT, TIME, TOD, DT, DATE, REAL, LREAL and STRING. There is also a special „type“ named INT1DP: It's an INT in the PLC, but in JavaScript the variable is of type float with 1 decimal place (i.e. a value of 568 in the PLC is 56.8 in JavaScript).

The library provides built-in conversion of date and time values to formatted strings and REAL values can be rounded to a desired number of decimal places. For writing arrays and arrays of structures there is an option to choose only one array item to send to the PLC instead of the whole array. Another feature is the automatic structure padding for exchanging data with ARM-based devices (i.e. CX90xx), which have a 4-byte data alignment. If the parameter „dataAlign4“ is set to „true“ you don't need to worry about the alignment of data structures.

This documentation is primarily intended for programmers, who are familiar with the basic concepts of the JavaScript language. I presume that the ADS WebService is already installed on the device you want to connect to. If not, visit http://infosys.beckhoff.com for more information about the installation of the server.

2. Instancing the Web Service Client

The heart of the library is the WebServiceClient constructor. You can create an instance by either using the „new“ keyword or the library function intented for this. Instancing the constructor you have to pass an object containing the arguments for the Web Service. Required are at least the URL of the TcAdsWebService.dll and the AMS-NetID of the PLC Runtime System. A minimal definition looks like this:

```
var PLC = TAME.WebServiceClient.createClient({
        serviceURL: "http://192.168.1.2/TcAdsWebService/TcAdsWebService.dll",
        amsNetId: "192.168.1.2.1.1"
});
```

Here is an example using all parameters :

```
var PLC = TAME.WebServiceClient.createClient({
        serviceURL: "http://192.168.1.2/TcAdsWebService/TcAdsWebService.dll",
        amsNetId: "192.168.1.2.1.1",
        amsPort: "802",
        dataAlign4: true,
        language: en
});
```

Most likely you will use a global variable for this, as you access the client from various functions.

After instancing the WebServiceClient you have an object („PLC“ in the above examples) wich provides the methods for sending read and write requests to the ADS WebService. They are described in the following chapters.

Parameters (Type, Required, Default)

serviceUrl (string, required, undefined)

The URL of the TcAdsWebservice.dll.

amsNetId (string, required, undefined)

The ADS-AMSNetID of the PLC.

amsPort (string, optional, **801**|802|803|804)

The ADS-Port number of the Runtime-System.

dataAlign4 (boolean, optional, undefined)

In ARM-based devices (i.e.CX90xx) the data in the memory is organised using a 4-byte data alignment. If structures are read/written from/to the PLC's memory, the data have to be aligned the same way. If they're not aligned you must either insert padding bytes manually or let the library do this for you by setting this parameter to „true“. Set this parameter only to „true“ if you exchange data with an ARM-based device.

language (string, optional, **ge**|en)

Set the language for names of days and months. This option is used for the formatted output of dates.

Instance Properties (Type, Default)

dateNames (object)

Contains an object with 4 arrays for the short and the full names of days and months used for the formatted output of dates. The language of the names can be set by the parameter „language“. If you need a language other then German or English, you can overwrite this property with your own names after instancing instead of changing the source code of the library.

maxStringLen (number, 255)

The maximum length of strings.

maxDropReq (number, 10)

The maximum number of dropped requests in conjunction with the „id“-parameter of the requests. When the number has been reached a new request is fired and the counter starts again.

useCheckBounds (boolean, true)

If set to „true“, the limits of numeric variables are checked before sending them to the PLC. If a limit is exceeded, the value will be set to the limit and an error message is written to the console. This applies to the following data types: BYTE, WORD , DWORD, USINT, SINT, UINT, INT, INT1DP, UDINT, DINT, TIME, REAL and LREAL.

3. The Basics: ReadReq and WriteReq

There are 2 basic methods of the WebServiceClient: The Read Request (readReq) for fetching data from the PLC and the Write Request (writeReq) for sending data to the PLC. Both functions have one paramter only: an object called Request Descriptor. They were the first methods implemented, all the other request functions are just a kind of shortcuts and call them passing an internally created request descriptor.

3.1 Read Request

Lets' use the instance of the WebServiceClient created in chapter 1. Assuming you want to read 3 PLC variables defined in a direct sequence, one of type integer at %MB53 and the other ones of type boolean at %MB55 and %MB56, a simple request definition looks like this:

```
PLC.readReq({
        addr: "%MB53",
        seq: true,
                items: [{
                        type: "INT",
                        jvar: "myVar1",
                },{
                        type: "BOOL",
                        jvar: "myVar2"
                },{
                        type: "BOOL",
                        jvar: "myVar3"
                }]
        }
);
```

If the variables aren't following each other in a direct sequence, you have to set the adress for each item separately. The request get's the whole data out of the PLC's memory, from the specified start address to the address of the last variable plus it's length. So it's a bad idea to leave too much space between the adresses.

```
PLC.readReq({
        addr: "%MB53",
                items: [{
                        addr: 53,
                        type: "INT",
                        jvar: "myVar1"
                },{
                        addr: 59,
                        type: "BOOL",
                        jvar: "myVar2"
                },{
                        addr: 60,
                        type: "BOOL",
                        jvar: "myVar3"
                }]
        }
);
```

As you can see, the boolean variables are set to „%MB“ addresses. You can't access multiple boolean variables with „%MX“ adresses. It's only possible to read one single variable this way. I recommend using the „readBool“ method for this.

Parameters (Type, Required, Default)

addr (string, required, undefined)

The start address of the data to be read.

debug (boolean, optional, false)

If this option is set to true, the request descriptor is written to the console.

id (number, optional, undefined)

All requests are completly independent of one another. If you have a request defined and call it through a loop, every execution generates an AJAX request of it's own. So if a requests definition is called in rapid succession one ore more requests can be fired before the previous one has finished. That doesn't make sense and can lead to stucking. This option is an ID to have a marker for the otherwise „anonymous“ requests invoked with the same definition. Be aware that you don't give different request definitions the same number. If this parameter is set to a value (> 0) and the request is fired, all following requests are dropped until the current one has finished or the number set in the „maxDropReq“ property of the webservice instance has been reached. For every dropped request a message is written to the console. This option makes only sense with fast cyclic reading or writing and has less meaning with modern browsers. Just try it for yourself.

items (array, required, undefined)

An array of objects containing the allocation of PLC and JavaScript variables (adresses, names and types).

oc (function, optional, undefined)

A function wich will be executed after the request has finished sucessfully.

ocd (number, optional, undefined)

A value of milliseconds for delaying the on-complete-function.

seq (boolean, optional, undefined)

If the variables in the PLC follow each other in a direct sequence, you don't have to specify the variable address of each one. You can omit the address parameters of the items and set this option to true instead.

Item Parameters (Type, Required, Default)

addr (number, required if the „seq“ parameter is false or undefined, undefined)

The address of the PLC variable to read.

type (string, required, undefined)

The type of the PLC variable. Can also contain a formatting part for some types. Look at chapter 7 for more information.

jvar (string, required, undefined)

The name of the JavaScript variable to store the data in. This must be a **global** variable. Namespaces of any depth and arrays are supported, i.e. "myNameSpace.myVars.var1" or "myNameSpace.myArray[0]".

prefix (string, optional, undefined)

An optional string wich can be added to the data. The data value will be converted to a string with the prefix inserted **before** the value.

suffix (string, optional, undefined)

An optional string wich can be added to the data. The data value will be converted to a string with the suffix inserted **after** the value.

3.2 Write Request

Sending values to the PLC ist also quite easy. The parameters of the request descriptor are almost the same as of the read request. But the adresses of the PLC variables have always to be in a direct sequence. In the example below 4 variables are written, beginning at MB105.

```
PLC.writeReq({
        addr: "%MB105",
                items: [{
                        type: "REAL",
                        val: 234.12
                },{
                        type: "STRING.10",
                        val: "Hello"
                },{
                        type: "BOOL",
                        val: myVar3
                },{
                        type: "BOOL",
                        val: false
                }]
        }
);
```

Note that you can't access multiple variables with „%MX“ adresses using this method. It's only possible to write one single variable this way. I recommend the „writeBool“ method for this.

Parameters (Type, Required, Default)

addr (string, required, undefined)

The start address of the data to be read.

debug (boolean, optional, false)

If this option is set to true, the request descriptor is written to the console.

id (number, optional, undefined)

All requests are completly independent of one another. If you have a request defined and call it through a loop, every execution generates an AJAX request of it's own. So if a requests definition is called in rapid succession one ore more requests can be fired before the previous one has finished. That doesn't make sense and can lead to stucking. This option is an ID to have a marker for the otherwise „anonymous“ requests invoked with the same definition. Be aware that you don't give different request definitions the same number. If this parameter is set to a value (> 0) and the request is fired, all following requests are dropped until the current one has finished or the number set in the „maxDropReq“ property of the webservice instance has been reached. For every dropped request a message is written to the console. This option makes only sense with fast cyclic reading or writing and has less meaning with modern browsers. Just try it for yourself.

items (array, required, undefined)

An array of objects containing the allocation of PLC and JavaScript variables (adresses, names and types).

oc (function, optional, undefined)

A function wich will be executed after the request has finished sucessfully.

ocd (number, optional, undefined)

A value of milliseconds for delaying the on-complete-function.

Item Parameters (Type, Required, Default)

type (string, required, undefined)

The type of the PLC variable. Can also contain a formatting part for some types. Look at chapter 7 for more information.

val (boolean|number|string, required, undefined)

The value to send to the PLC. Can also be specified as a variable (even a local definend one).

4. Type Dependent Requests

For accessing single PLC variables, the library provides read and write requests for each supported data type.

4.1 Read Requests

Except for some special formatting options, the API of the requests is the same for all data types.

An example of reading a boolean variable:

```
PLC.readBool({
        addr: "%MB53",
        jvar: "myVar"
});
```

I have tried to keep the syntax short, so it's no problem to write the commands in a single line:

```
PLC.readBool({addr: "%MB53", jvar: "myVar"});
```

An example of reading a string defined with 10 characters:

```
PLC.readString({
        addr: "%MB102",
        strlen: 10,
        jvar: "myVar"
});
```

An example of reading a PLC REAL value with rounding it to 1 decimal place. 100 ms after completing the request an alert window will be opened for displaying the value.

```
PLC.readReal({
        addr: "%MB240",
        dp: 1,
        jvar: "myVar",
        ocd: 100,
        oc: function() {
                alert("Value: " + myVar);
        }
});
```

With the „readBool“- method you can also access bit values with %MX-addresses.

```
PLC.readBool({
        addr: "%MX13.1",
        jvar: "myVar"
});
```

Supported Data Types

PLC Data Type	Method	Note
BOOL	readBool	
BYTE	readByte	Basically the same as „readUsint“.
WORD	readWord	Basically the same as „readUint“.
DWORD	readDword	Basically the same as „readUdint“.
USINT	readUsint	
SINT	readSint	
UINT	readUint	
INT	readInt	
INT	readInt1Dp	An INT converted to a variable with 1 decimal place, i.e. 637 in the PLC is 63.7 in the JavaScript variable.
UDINT	readUdint	
DINT	readDint	
TIME	readTime	With the „format“-parameter set the time value is converted to a formatted string, otherwise the output is a string with a value in milliseconds.
TOD	readTod	With the „format“-parameter set the time value is converted to a formatted string, otherwise the output is a date object.
DT	readDt	With the „format“-parameter set the time value is converted to a formatted string, otherwise the output is a date object.
DATE	readDate	With the „format“-parameter set the time value is converted to a formatted string, otherwise the output is a date object.
REAL	readReal	Has a special formatting parameter „dp“ to set the number of the decimal places. For issues with converting REAL variables see chapter 8.1.
LREAL	readLreal	Has a special formatting parameter „dp“ to set the number of the decimal places.
STRING	readString	Has a special parameter „strlen“ to set the length of the string.

Common Parameters (Type, Required, Default)

addr (string, required, undefined)

The address of the variable to be read.

debug (boolean, optional, false)

If this option is set to true, the request descriptor is written to the console.

id (number, optional, undefined)

All requests are completly independent of one another. If you have a request defined and call it through a loop, every execution generates an AJAX request of it's own. So if a requests definition is called in rapid succession one ore more requests can be fired before the previous one has finished. That doesn't make sense and can lead to stucking. This option is an ID to have a marker for the otherwise „anonymous“ requests invoked with the same definition. Be aware that you don't give different request definitions the same number. If this parameter is set to a value (> 0) and the request is fired, all following requests are dropped until the current one has finished or the number set in the „maxDropReq“ property of the webservice instance has been reached. For every dropped request a message is written to the console. This option makes only sense with fast cyclic reading or writing and has less meaning with modern browsers. Just try it for yourself.

jvar (string, required, undefined)

The name of the JavaScript variable to store the data in. This must be a **global** variable. Namespaces of any depth and arrays are supported, i.e. "myNameSpace.myVars.var1" or "myNameSpace.myArray[0]".

oc (function, optional, undefined)

A function wich will be executed after the request has finished sucessfully.

ocd (number, optional, undefined)

A value of milliseconds for delaying the on-complete-function.

prefix (string, optional, undefined)

An optional string wich can be added to the data. The data value will be converted to a string with the prefix inserted **before** the value.

suffix (string, optional, undefined)

An optional string wich can be added to the data. The data value will be converted to a string with the suffix inserted **after** the value.

Type Dependent Parameters (Type, Required, Default)

dp | decPlaces (number, optional, undefined)

For data types REAL and LREAL: The number of decimal places. For issues with converting REAL variables see chapter 8.1.

format (string, optional, undefined)

For data types TIME, TOD, DT, DATE: A string containing information for the formatted output of date and/or time. If used, the output ist a string instead of a date object. See chapters 8.3 and 8.4 for this.

strlen (number, required if string length <> default, 80)

For data type STRING: The defined lenght of the string in the PLC.

4.2 Write Requests

Except for the length parameter of strings, the API of the requests is the same for all data types. An example of writing an integer:

```
PLC.writeInt({
        addr: "%MB106",
         val: 2635
});
```

Set a %MX-addressed PLC variable to „true“ and 1 second later to „false“:

```
PLC.writeBool({
        addr: "%MX110.4",
        val: true,
        ocd: 1000,
        oc: function() {
                PLC.writeBool({addr: "%MX110.4", val: false});
        }
});
```

Writing a string defined with 20 characters:

```
PLC.writeString({
        addr: "%MB260",
        strlen: 20,
        val: "Hello!"
});
```

Supported Data Types

PLC Data Type	Method	Note
BOOL	writeBool	
BYTE	writeByte	Basically the same as „writeUsint“.
WORD	writeWord	Basically the same as „writeUint“.
DWORD	writeDword	Basically the same as „writeUdint“.
USINT	writeUsint	
SINT	writeSint	
UINT	writeUint	
INT	writeInt	
INT	writeInt1Dp	The JavaScript variable is multiplied by 10 and converted to an integer, i.e. a value of 12.313 in JS is 123 in the PLC.
UDINT	writeUdint	
DINT	writeDint	
TIME	writeTime	Has a special parameter „format“. See chapter 8.3 for this.
TOD	writeTod	The value has to be a JS date object.
DT	writeDt	The value has to be a JS date object.
DATE	writeDate	The value has to be a JS date object.
REAL	writeReal	For issues with converting REAL variables see chapter 8.1.
LREAL	writeLreal	
STRING	writeString	Has a special parameter „strlen“ to set the length of the string.

Common Parameters (Type, Required, Default)

addr (string, required, undefined)

The start address of the data to be written.

debug (boolean, optional, false)

If this option is set to true, the request descriptor is written to the console.

id (number, optional, undefined)

All requests are completly independent of one another. If you have a request defined and call it through a loop, every execution generates an AJAX request of it's own. So if a requests definition is called in rapid succession one ore more requests can be fired before the previous one has finished. That doesn't make sense and can lead to stucking. This option is an ID to have a marker for the otherwise „anonymous“ requests invoked with the same definition. Be aware that you don't give different request definitions the same number. If this parameter is set to a value (> 0) and the request is fired, all following requests are dropped until the current one has finished or the number set in the „maxDropReq“ property of the webservice instance has been reached. For every dropped request a message is written to the console. This option makes only sense with fast cyclic reading or writing and has less meaning with modern browsers. Just try it for yourself.

oc (function, optional, undefined)

A function wich will be executed after the request has finished sucessfully.

ocd (number, optional, undefined)

A value of milliseconds for delaying the on-complete-function.

val (boolean|number|string, required, undefined)

The value to send to the PLC. Can also be specified as a variable (even a local definend one).

Type Dependent Parameters (Type, Required, Default)

strlen (number, required if string length <> default, 80)

For data type STRING: The defined lenght of the string in the PLC.

5. Arrays

There are some powerful methods for accessing arrays and structures in the PLC. This way you can read or write thousands of values with one single command. Just like for the single variables, there are reading and writing methods for each data type.

5.1 Read Requests

An example of reading an array of 5 strings, each defined for 10 characters length:

```
PLC.readArrayOfString({
        addr: "%MB110",
        arrlen: 5,
        strlen: 10,
        jvar: "myArray"
});
```

An example of reading an array of 10 DATE values, each defined for 10 characters length. After executing, the JavaScript array will contain formatted strings.

```
PLC.readArrayOfDate({
        addr: "%MB10",
        arrlen: 10,
        format: "#WEEKDAY#, #DD#.#MM#.#YYYY",
        jvar: "myArray"
});
```

Supported Data Types

PLC Data Type	Method	Note
BOOL	readArrayOfBool	
BYTE	readArrayOfByte	Basically the same as „readArrayOfUsint“.
WORD	readArrayOfWord	Basically the same as „readArrayOfUint“.
DWORD	readArrayOfDword	Basically the same as „readArrayOfUdint“.
USINT	readArrayOfUsint	
SINT	readArrayOfSint	
UINT	readArrayOfUint	
INT	readArrayOfInt	
INT	readArrayOfInt1Dp	An INT converted to a variable with 1 decimal place, i.e. 637 in the PLC is 63.7 in the JavaScript variable.
UDINT	readArrayOfUdint	
DINT	readArrayOfDint	
TIME	readArrayOfTime	With the „format“-parameter set the time value is converted to a formatted string, otherwise the output is a string with a value in milliseconds.
TOD	readArrayOfTod	With the „format“-parameter set the time value is converted to a formatted string, otherwise the output is a date object.
DT	readArrayOfDt	With the „format“-parameter set the time value is converted to a formatted string, otherwise the output is a date object.
DATE	readArrayOfDate	With the „format“-parameter set the time value is converted to a formatted string, otherwise the output is a date object.
REAL	readArrayOfReal	Has a special formatting parameter „dp“ to set the number of the decimal places. For issues with converting REAL variables see chapter 8.1.
LREAL	readArrayOfLreal	Has a special formatting parameter „dp“ to set the number of the decimal places.

STRING	readArrayOfString	Has a special parameter „strlen“ to set the length of the string.

Common Parameters (Type, Required, Default)

addr (string, required, undefined)

The address of the variable to be read.

arrlen (number, required, undefined)

The length of the array (the number of the items).

debug (boolean, optional, false)

If this option is set to true, the request descriptor is written to the console.

id (number, optional, undefined)

All requests are completly independent of one another. If you have a request defined and call it through a loop, every execution generates an AJAX request of it's own. So if a requests definition is called in rapid succession one ore more requests can be fired before the previous one has finished. That doesn't make sense and can lead to stucking. This option is an ID to have a marker for the otherwise „anonymous“ requests invoked with the same definition. Be aware that you don't give different request definitions the same number. If this parameter is set to a value (> 0) and the request is fired, all following requests are dropped until the current one has finished or the number set in the „maxDropReq“ property of the webservice instance has been reached. For every dropped request a message is written to the console. This option makes only sense with fast cyclic reading or writing and has less meaning with modern browsers. Just try it for yourself.

jvar (string, required, undefined)

The name of the JavaScript variable to store the data in. This must be a **global** variable. Namespaces of any depth and arrays are supported, i.e. "myNameSpace.myVars.var1" or "myNameSpace.myArray[0]".

oc (function, optional, undefined)

A function wich will be executed after the request has finished sucessfully.

ocd (number, optional, undefined)

A value of milliseconds for delaying the on-complete-function.

Type Dependent Parameters (Type, Required, Default)

dp | decPlaces (number, optional, undefined)

For data type REAL: The number of decimal places. For issues with converting REAL variables see chapter 8.1.

format (string, optional, undefined)

For data types TIME, TOD, DT, DATE: A string containing information for the formatted output of date and/or time. If used, the output ist a string instead of a date object.

strlen (number, required if string length <> default, 80)

For data type STRING: The defined lenght of the string in the PLC.

5.2 Write Requests

An example of writing an array of 5 boolean variables:

```
PLC.writeArrayOfBool({
        addr: "%MB110",
        arrlen: 5,
        val: myArray
});
```

You can also write only one item of the array. Here an example of writing the last element of an array of 10 integer variables:

```
PLC.writeArrayOfInt({
        addr: "%MB230",
        arrlen: 10,
        item: 9,
        val: myArray
});
```

Supported Data Types

PLC Data Type	Method	Note
BOOL	writeArrayOfBool	
BYTE	writeArrayOfByte	Basically the same as „writeArrayOfUsint“.
WORD	writeArrayOfWord	Basically the same as „writeArrayOfUint“.
DWORD	writeArrayOfDword	Basically the same as „writeArrayOfUdint“.
USINT	writeArrayOfUsint	
SINT	writeArrayOfSint	
UINT	writeArrayOfUint	
INT	writeArrayOfInt	
INT	writeArrayOfInt1Dp	The JavaScript variables are multiplied by 10 and converted to an integer, i.e. a value of 12.313 in JS is 123 in the PLC.
UDINT	writeArrayOfUdint	
DINT	writeArrayOfDint	
TIME	writeArrayOfTime	Has a special parameter „format“. See chapter 8.3 for this.
TOD	writeArrayOfTod	The values have to be JS date objects.
DT	writeArrayOfDt	The values have to be JS date objects.
DATE	writeArrayOfDate	The values have to be JS date objects.
REAL	writeArrayOfReal	For issues with converting REAL variables see chapter 8.1.
LREAL	writeArrayOfLreal	
STRING	writeArrayOfString	Has a special parameter „strlen“ to set the length of the string.

Common Parameters (Type, Required, Default)

addr (string, required, undefined)

The start address of the data to be written.

arrlen (number, optional, undefined)

The length of the array (the number of the items). Will be detected by TAME if not set.

debug (boolean, optional, false)

If this option is set to true, the request descriptor is written to the console.

id (number, optional, undefined)

All requests are completly independent of one another. If you have a request defined and call it through a loop, every execution generates an AJAX request of it's own. So if a requests definition is called in rapid succession one ore more requests can be fired before the previous one has finished. That doesn't make sense and can lead to stucking. This option is an ID to have a marker for the otherwise „anonymous“ requests invoked with the same definition. Be aware that you don't give different request definitions the same number. If this parameter is set to a value (> 0) and the request is fired, all following requests are dropped until the current one has finished or the number set in the „maxDropReq“ property of the webservice instance has been reached. For every dropped request a message is written to the console. This option makes only sense with fast cyclic reading or writing and has less meaning with modern browsers. Just try it for yourself.

item (number, optional, undefined)

If this parameter is set to the index number (starting with 0) of an array item, only this item will be written to the PLC.

oc (function, optional, undefined)

A function wich will be executed after the request has finished sucessfully.

ocd (number, optional, undefined)

A value of milliseconds for delaying the on-complete-function.

val (array, required, undefined)

The array of values to send to the PLC.

Type Dependent Parameters (Type, Required, Default)

strlen (number, required if string length <> default, 80)
For data type STRING: The defined lenght of the string in the PLC.

6. Structures

Accessing structures is somewhat more complex. The request needs information about the elements of the structure in the PLC's memory. For this the descriptor has the „def“ parameter. It requires an object literal containing the names of the propertys of the storing JavaScript object and the corresponding data types.

Assuming you have a user defined data type like this:

```
TYPE ST_Test :
	STRUCT
		nByte: BYTE;
		fReal: REAL;
		iSint: SINT;
		tTime: TIME;
		arrString: ARRAY[1..5] OF STRING(6);
		iInt: INT;
	END_STRUCT
END_TYPE
```

And a variable of this type with the address %MB500:

```
TestStruct AT%MB500:  ST_Test;
```

The structure definition could look like this. You can set the names of the properties as you need. The order of the data types has to be the same as in the PLC.

```
var structdef = {
        var_1: "BYTE",
        var_2: "REAL.2",
        var_3: "SINT",
        var_4: "TIME",
        var_5: "ARRAY.5.STRING.6",  //Array of 5 strings with 6 chars
        var_6: "INT1DP"
};
```

Note that you can use one definition for read and write requests. If used with a write request, all formatting options will be ignored, except of the parameter for the length of strings.

6.1 Read Request

Of course you have to define a JavaScript object to store the data, the properties of the object contain the values. For read requests normally you haven't to define the properties, they will be created if the request is executed. So instead of:

```
var myVar = {
        var_1: 0,
        var_2: 0,
        var_3: 0,
        var_4: new Date(),
        var_5: [],
        var_6: 0
};
```

You could define the object just like this:

```
var myVar = {};
```

But if you have an array in the structure, this property must be defined:

```
var myVar = {var_5: []};
```

The request again is quite simple:

```
PLC.readStruct({
        addr: "%MB500",
        def: structdef,
        jvar: "myVar"
});
```

Parameters (Type, Required, Default)

addr (string, required, undefined)

The address of the variable to be read.

debug (boolean, optional, false)

If this option is set to true, the request descriptor is written to the console.

def (object, required, undefined)

An object containing information about the structure definition.

id (number, optional, undefined)

All requests are completly independent of one another. If you have a request defined and call it through a loop, every execution generates an AJAX request of it's own. So if a requests definition is called in rapid succession one ore more requests can be fired before the previous one has finished. That doesn't make sense and can lead to stucking. This option is an ID to have a marker for the otherwise „anonymous“ requests invoked with the same definition. Be aware that you don't give different request definitions the same number. If this parameter is set to a value (> 0) and the request is fired, all following requests are dropped until the current one has finished or the number set in the „maxDropReq“ property of the webservice instance has been reached. For every dropped request a message is written to the console. This option makes only sense with fast cyclic reading or writing and has less meaning with modern browsers. Just try it for yourself.

jvar (string, required, undefined)

The name of the JavaScript variable to store the data in. This must be a **global** variable. Namespaces of any depth and arrays are supported, i.e. "myNameSpace.myVars.var1" or "myNameSpace.myArray[0]".

oc (function, optional, undefined)

A function wich will be executed after the request has finished sucessfully.

ocd (number, optional, undefined)

A value of milliseconds for delaying the on-complete-function.

6.2 Write Request

For write requestes applies the same as for read requests. The difference is, of course, that the JavaScript object and it's properties must exist befor the request is executed.

```
var myVar = {
        var_1: 12,
        var_2: 7464.313,
        var_3: -96,
        var_4: new Date("January 3, 2012 18:54:00"),
        var_5: ["This", "is", "a", "string"," array"],
        var_6: 34.5
};
```

Like the read request, the write request is also quite simple:

```
PLC.writeStruct({
        addr: "%MB500",
        def: structdef,
        val: myVar
});
```

Parameters (Type, Required, Default)

addr (string, required, undefined)

The start address of the data to be written.

debug (boolean, optional, false)

If this option is set to true, the request descriptor is written to the console.

def (object, required, undefined)

An object containing information about the structure definition.

id (number, optional, undefined)

All requests are completly independent of one another. If you have a request defined and call it through a loop, every execution generates an AJAX request of it's own. So if a requests definition is called in rapid succession one ore more requests can be fired before the previous one has finished. That doesn't make sense and can lead to stucking. This option is an ID to have a marker for the otherwise „anonymous“ requests invoked with the same definition. Be aware that you don't give different request definitions the same number. If this parameter is set to a value (> 0) and the request is fired, all following requests are dropped until the current one has finished or the number set in the „maxDropReq“ property of the webservice instance has been reached. For every dropped request a message is written to the console. This option makes only sense with fast cyclic reading or writing and has less meaning with modern browsers. Just try it for yourself.

oc (function, optional, undefined)

A function wich will be executed after the request has finished sucessfully.

ocd (number, optional, undefined)

A value of milliseconds for delaying the on-complete-function.

val (object, required, undefined)

The array of values to send to the PLC.

7. Arrays Of Structures

The most powerful method's of the library can read or write a whole array of structures. They are a combination of the methods for accessing arrays and structures and you should read the previous chapters to understand whats going on. Based on the structure definition of chapter 5 here is the definition of a PLC variable :

```
TestArray AT%MB500:  ARRAY [1..2] OF ST_Test;
```

7.1 Read Request

The definition of the JavaScript object. At least the properties containing arrays must be defined.

```
var myArray = [{
        var_5: []
},{
        var_5: []
}];
```

The request definintion itself is quite simple again.

```
PLC.readArrayOfStruct({
        addr: "%MB500",
        def: structdef,
        arrlen: 2,
        jvar: "myArray"
});
```

Parameters (Type, Required, Default)

addr (string, required, undefined)

The address of the variable to be read.

arrlen (number, required, undefined)

The length of the array (the number of the items).

debug (boolean, optional, false)

If this option is set to true, the request descriptor is written to the console.

def (object, required, undefined)

An object containing information about the structure definition.

id (number, optional, undefined)

All requests are completly independent of one another. If you have a request defined and call it through a loop, every execution generates an AJAX request of it's own. So if a requests definition is called in rapid succession one ore more requests can be fired before the previous one has finished. That doesn't make sense and can lead to stucking. This option is an ID to have a marker for the otherwise „anonymous“ requests invoked with the same definition. Be aware that you don't give different request definitions the same number. If this parameter is set to a value (> 0) and the request is fired, all following requests are dropped until the current one has finished or the number set in the „maxDropReq“ property of the webservice instance has been reached. For every dropped request a message is written to the console. This option makes only sense with fast cyclic reading or writing and has less meaning with modern browsers. Just try it for yourself.

jvar (string, required, undefined)

The name of the JavaScript variable to store the data in. This must be a **global** variable. Namespaces of any depth and arrays are supported, i.e. "myNameSpace.myVars.var1" or "myNameSpace.myArray[0]".

oc (function, optional, undefined)

A function wich will be executed after the request has finished sucessfully.

ocd (number, optional, undefined)

A value of milliseconds for delaying the on-complete-function.

7.2 Write Request

```
var myArray = [{
        var_1: 12,
        var_2: 7464.313,
        var_3: -96,
        var_4: new Date("January 3, 2012 18:54:00"),
        var_5: ["This", "is", "a", "string", " array."],
        var_6: 34.5
},{
        var_1: 109,
        var_2: 873.40,
        var_3: -5,
        var_4: new Date(1325619660),
        var_5: ["This", "is a",  "string", " array", "too." ],
        var_6: 122.9
}];
```

```
PLC.writeArrayOfStruct({
        addr: "%MB500",
        def: structdef,
        arrlen: 2,
        val: myArray
});
```

Note that you also can use the „item“-parameter, if you want to write a single array item (one structure) only.

Parameters (Type, Required, Default)

addr (string, required, undefined)

The start address of the data to be written.

arrlen (number, optional, undefined)

The length of the array (the number of the items). Will be detected by TAME if not set.

debug (boolean, optional, false)

If this option is set to true, the request descriptor is written to the console.

def (object, required, undefined)

An object containing information about the structure definition.

id (number, optional, undefined)

All requests are completly independent of one another. If you have a request defined and call it through a loop, every execution generates an AJAX request of it's own. So if a requests definition is called in rapid succession one ore more requests can be fired before the previous one has finished. That doesn't make sense and can lead to stucking. This option is an ID to have a marker for the otherwise „anonymous“ requests invoked with the same definition. Be aware that you don't give different request definitions the same number. If this parameter is set to a value (> 0) and the request is fired, all following requests are dropped until the current one has finished or the number set in the „maxDropReq“ property of the webservice instance has been reached. For every dropped request a message is written to the console. This option makes only sense with fast cyclic reading or writing and has less meaning with modern browsers. Just try it for yourself.

item (number, optional, undefined)

If this parameter is set to the index number (starting with 0) of an array item, only this item will be written to the PLC.

oc (function, optional, undefined)

A function wich will be executed after the request has finished sucessfully.

ocd (number, optional, undefined)

A value of milliseconds for delaying the on-complete-function.

val (array, required, undefined)

The array of values to send to the PLC.

8. Type Dependent Parameters

8.1 REAL and LREAL

The data type REAL is a 32 bit numeric variable with one bit as algebraic sign, 8 bit for the exponent and 23 bit for the mantissa. Though a decimal floating point number can be converted to a binary floating point number, not every value can be represented due to the limited number of digits in the mantissa. In example, the decimal floating point number 0,1 will be converted to a value of 0.10000000149011612. Moreover, the data type FLOAT in JavaScript has a lenght of 64 bit but the type REAL in the PLC has a length of 32 bit.

For reading PLC REAL values there is a parameter for rounding the value to the desired number of places after the decimal point. I recommend to use it always. For writing REAL variables I have tried to find a fast and also accurate conversion, but I cannot rule out the possibility to get weird values while using those methods.

For the method „readReq“ and those for reading structures just add the number of decimal places to the type, separated by a point:

```
PLC.readReq({
        addr: "%MB53",
        seq: true,
                items: [{
                        type: "REAL.1",
                        jvar: "myVar"
                }]
        }
);
```

Example of a structure definition for „readStruct“ and „readArrayofStruct“:

```
var structdef = {
        myreal: "REAL.2"
};
```

For the methods „readReal“ and „readArrayOfReal“ there is the seperate parameter „dp“ or „decPlaces“:

```
PLC.readReal({
        addr: "%MB240",
        dp: 2,
        jvar: "myVar"
});
```

The data type LREAL is a 64 bit numeric variable with one bit as algebraic sign, 11 bit for the exponent and 52 bit for the mantissa. Like the type REAL it can not represent every value of a decimal floating point number but it is much more precise. And there are no rounding errors when converting the values, cause both sides use the same format and length for storing the values in the memory. The parameter for rounding the value to the desired number of places after the decimal point is also supported.

8.2 STRING

A string in TwinCAT has a lenght of 80 characters by default. But you can change this and specify the length from 1 to 255 characters if you need.

This is a definition of a string with 10 characters length:

```
MyString AT %MB16: STRING(10);
```

If you do so, you have to set the length in your JavaScript code too, both for read and for write requests. For the methods „readReq“, „writeReq“ and those for reading/writing structures just add the number to the type separated by a point:

```
PLC.readReq({
        addr: "%MB16",
        seq: true,
                items: [{
                        type: "STRING.10",
                        jvar: "myVar"
                }]
        }
);
```

Example of a structure definition for „readStruct“ and „readArrayofStruct“:

```
var structdef = {
        mystring: "STRING.10"
};
```

For the methods „readString“, „writeString“, „readArrayOfString“ and „writeArrayOfString“ there is a seperate parameter named „strlen“:

```
PLC.writeString({
        addr: "%MB16",
        strlen: 10,
        val: "Hello world!"
});
```

There is another thing you have to be aware of when you work with strings. Strings in TwinCAT are null-terminated, so the real length in memory is the specified length + 1 byte (11 bytes for the examples above, 81 bytes for a standard string). The library adds the termination automatically, so you don't need to worry about it. But you must pay attention when you set the addresses for string variables.

8.3 TIME

When you read a TIME value from the PLC you will get a string with a value in milliseconds, if no formatting parameter is used. For the methods „readReq“, „writeReq“ and those for reading/writing structures just add the formatting string to the type separated by a point, for type dependent requests use the „format“-parameter.

The separator for placeholders and custom characters/substrings is the „#“-character. At this separator the parser splits the string into pieces and searches for formatting instructions.

Example of a read request, it returns a value converted to hours. A value of 90 minutes in the PLC would be a value of "1.5" in the JavaScript variable.

```
PLC.readReq({
        addr: "%MB50",
        seq: true,
                items: [{
                        type: "TIME.#h",
                        jvar: "myVar"
                }]
        }
);
```

For read requests you can use a combination of formatting instructions. A value of 92 minutes and 22 seconds would be a string of "1 - 32 - 22" in the JavaScript variable with the example below..

```
var structdef = {
        mytime: "TIME.#hh# - #mm# - #ss"
};
```

I've tried to keep the formatting flexible, so you can build whole sentences.

```
var structdef = {
        mytime: "TIME.#Actual time: #hh# hours, #mm# minutes and #ss# seconds."
};
```

For writing values to a PLC TIME variable without formatting you also have to use a value in milliseconds. But other then the write request you can use only one formatting instruction (i.e. minutes or hours).

An example of writing a value of 1200 ms:

```
PLC.writeTime({
        addr: "%MB50",
        val: 1200
});
```

Writing a value of 4 minutes:

```
PLC.writeTime({
        addr: "%MB50",
        format: #m,
        val: 4
});
```

Writing a value of 30 minutes with formatting in hours:

```
PLC.writeTime({
        addr: "%MB50",
                items: [{
                        type: "TIME.#h",
                        val: 0.5
                }]
        }
);
```

Formatting Instructions

Instruction	Return Value
none	milliseconds
dd	days, if the value is < 10 a leading zero will be added (i.e. 06)
d	days
hh	hours, if the value is < 10 a leading zero will be added.
h	hours
mm	minutes, if the value is < 10 a leading zero will be added.
m	minutes
ss	seconds, if the value is < 10 a leading zero will be added.
s	seconds
msmsms	milliseconds, 3 digits
ms	milliseconds

8.4 TOD, DT and DATE

Altough the parameters and instructions are almost the same as of type TIME, the conversion is a bit different. For example, when you read a TOD value of 0 hours and 30 minutes from the PLC using a „#h“ formatting instruction, you will get a value of 0, with the type TIME it would be 0.5. If no formatting string is used, the methods for these three types return JavaScript date objects.

```
PLC.readReq({
        addr: "%MB250",
        seq: true,
                items: [{
                        type: "TOD.#hh#:#mm",
                        jvar: "myVar"
                }]
        }
);
```

A read request with 2 variables:

```
PLC.readReq({
        addr: "%MB260",
        seq: true,
                items: [{
                        type: "DT.#WEEKDAY#, #hh#:#mm",
                        jvar: "myVar"
                },{
                        type: "DATE.#DD#. #MONTH# #YYYY",
                        jvar: "myVar2"
                }]
        }
);
```

An example of reading an array of 5 DT values:

```
PLC.readArrayOfDt({
        addr: "%MB120",
        arrlen: 5,
        format: "#DD#.#MM#.#YY#, #hh#:#mm",
        jvar: "myArray"
});
```

The write requests for these types require a date object as input value. Formatting parameters are not supported.

```
var myVar = new Date("January 3, 2012 18:54:00");

PLC.writeDt({
        addr: "%MB120",
        val: myVar
});
```

Formatting Instructions

Instruction	Return Value
none	JavaScript date object
YYYY	full year
YY	year
MONTH	full name of month
MON	short name of month
MM	month 01-12, 2 digits
M	month 1-12, 1or 2 digits
WEEKDAY	weekday, full name
WKD	weekday, short name
WD	weekday 0-6
DD	day of month 01-31, 2 digits
D	day of month 1-31, 1 or 2 digits
hh	hours, if the value is < 10 a leading zero will be added.
h	hours
mm	minutes, if the value is < 10 a leading zero will be added.
m	minutes
ss	seconds, if the value is < 10 a leading zero will be added.
s	seconds
msmsms	milliseconds, up to 2 leading zeros will be added
ms	milliseconds

9. Debugging

- If you encounter any problems, you should have a look at the JavaScript console of your browser. The library generates various messages if something goes wrong. Search for the term „TAME library error:“ and read the text after it. Maybe you will get a hint what kind of problem occured. In many cases the following lines will contain additional information like values or objects.
- You can also use the „debug“ parameter that every request supports. If you execute a request with this option set to „true“ the generated request descriptor will be written to the console.
- During the development of a project you should use the TAME version with comments, not the minified one as error messages can be crippled.
- Don't forget that JavaScript is case sensitive.

10. Tips & Tricks

- For a good performance it is important that not too many requests are executed at the same time. I recommend to have only one request for fast cyclic polling (<= 1s).
- Updating the DOM of the HTML page is far more resource intensive then fetching and parsing values. To get an idea: Parsing 2000 variables (DT and INT) takes about 20 ms with a Core-i5 PC and Firefox 10, an older PIII 2,4 GHz PC needs 50 ms. So you can read a large amount of variables with a cyclic request and update only the visible elements.
- If you encounter stuckings with fast cyclic polling on slow connections you can set an ID for the request and look if it gets better.
- Other than x86-based systems (i.e. PC's, CX10xx) ARM-based devices (i.e. CX90xx) use a 4-byte memory alignment. Every variable of 4-byte length has to be set to an address that must be divisible by 4 (compiling the project TwinCAT otherwise generates an error message).

  When defining data structures the 4-byte variables should come first, then the 2-byte varibles and at least the ones of 1 byte length. But you don't need to worry about that if you set the parameter „dataAlign4“ to „true“ (see Chapter 2). In this case TAME will generate padding bytes to match the alignment of the structure in the PLC's memory.
- The domain of the URL of your browser based visualisation and the one of the WebService's URL set in the parameter „serviceUrl“ of the WebServiceClient (see Chapter 2) have to be the same. Web browsers have built in a thing called „same-origin-policy“ wich prevents the access to content from other domains. If you want to access the WebService across multiple domains (i.e. a from a LAN and also from the outside over DynDNS) you normally have to set 2 or more client definitions. To circumvent this you can use the property „location“ of the JavaScript window object.

  So instead of using a fixed domain:

  „serviceUrl: 'http://192.168.1.2/TcAdsWebService/TcAdsWebService.dll',“,

  better use „window.location“:

  „serviceUrl: window.location.protocol + "//" + window.location.hostname + '/TcAdsWebService/TcAdsWebService.dll',“,

  or, if you use only Webkit based browsers:

  „serviceUrl: window.location.origin + '/TcAdsWebService/TcAdsWebService.dll',“.