www.asix.com.pl

Scripts

AsScripter Module

Doc. No. ENP8026
Version: 2016-02-23

ASKOM

ASKOM Sp. z o. o., ul. Józefa Sowińskiego 13, 44-121 Gliwice,
tel. +48 32 3018100, fax +48 32 3018101,
http://www.askom.com.pl, e-mail: office@askom.com.pl

Table of Contents

1 General Remarks

The purpose of the module is to expand the set of functions of the Asix system with:

- execution of non-standard calculations on process variable values;
- programming non-standard responses to events within the Asix module;
- transferring the variable values and other information on the Asix system outside;
- data acquisition from non-standard data sources.

In order to perform the above-mentioned tasks, the script module enables the following:

- automatic activation of many concurrent scripts created in different programming languages with the ability to define the script priority/validity;
- data exchange between scripts;
- calling from given script the functions included in another script created in various languages (library scripts);
- passing the control to the Asix system as a result of events arisen during the script execution (operator actions);
- control of script execution time;
- tracing changes in a file containing the script program and automatic restarting the script after detection of such changes.

2 Program Requirements

The script module is designed basing on ActiveX ® Scripting technology and can execute scripts created in languages, which interpreters use this technology. The script module was tested with use of Microsoft Windows Script 5.5, which contains JScript 5.5 and VBScript 5.5 interpreters.

The script module can be used in Microsoft Windows NT 4.0, Windows 2000, Windows XP, Windows 98, and Windows ME. If Windows Script is not installed, it can be found at http://www.microsoft.com/msdownload/vbscript/scripting.asp.

Microsoft Script Debugger, which can be used for scripts debugging, can be found at http://msdn.microsoft.com/scripting.

In Asix, the script module operates in version 3 or higher.

3 Start-up and Execution of Scripts

Scripts can be started up automatically by the script module as a result of appropriate items in the application's initialization file or by the SCRIPT operator action.

Declaration of scripts is realized by means of Architect:

Architect > *Fields and Computers* > *Scripts and actions* module > *Scripts* tab

Scripts tab enables declaration of scripts extending the functional abilities of Asix system. Scripts must be declared in the subsequent items, giving the script name and declaring the file and additional script options according to the syntax described below.

Script name must be different than the names of other parameters declaring the usage of script. Script name cannot be the name of script module parameter and cannot be the word “skrypt" or “script". At the moment, the script module recognizes the VBScript (.vbs) and JScript (.js) language extensions. Declaration of script file with additional options requires the following syntax:

script_file script_parameters executive_parameters

where:
script_file - path and file name containing the script program; file name extension
defines the default script name;
script_parameters - parameters passed to the script; script parameters are available in the
script with use of Asix.Script.Parameters object (or Asix.Script.Arguments object);
executive_parameters - parameters passed to the scripting module (not to the script) and determining the script execution parameters, which are different than the default ones;

The executive parameters include: limitation of execution time, debugger activation, declaration of thread, where the script will be executed and its priority, script type, language etc. Some parameters (e.g. limitations) are similar to scripting module parameters. They are placed as executive parameters only when the defined script has special requirements, different than the ones defined with use of parameters defining the script execution method – see: *Scripting parameters* tab.

Executive parameters are given in convention of cscript program, i.e. they are preceded with double slash.

Executive parameters:
//S - in the given moment, only one script with parameter //S, included in the given file can run; parameter is important only when script is launched as Asix operator action; scripting module will reject the next attempt of script

launch in situation, when the previous script instance with parameter //S included in the same file is not completed yet; if the parameter is missing, it is possible to launch at the same time multiple scripts based on program included in the same file;

//X - when the script is launched, a debugger (installed in the system as the default one) is also started; debugger stops the script execution on the first script instruction;

//D - if error in script is found, debugger will be started;

//IT:nnn - limitation of initialization part execution time, expressed in milliseconds;

//T:nnn - limitation of event part execution time, expressed in milliseconds;

//E:name - gives the script language name (e.g. //E:JScript);

//Thread:name[,pri] - gives the thread name, where the script is executed and defines pri priority of the thread; thread name can be custom and is used to group the scripts executed in one thread; script thread priorities:
0 IDLE,
1 LOWEST,
2 BELOW_NORMAL,
3 NORMAL - (default),
4 ABOVE_NORMAL,
5 HIGHEST,
6 TIME_CRITICAL;
default thread priority is priority 3 (normal).

//U - if this parameter is given, “OnRead” events and temporal event functions defined by SetInterval and ExecuteAt methods will not be generated, if Asix application windows are hidden (e.g. with use of HIDE operator action).

3.1 Threads

As scripts can perform tasks with various time requirements from calculating the values of frequently read process variables through periodical report generation, script module enables to group scripts with similar time requirements and similar importance level.

Every script is executed in the context of a thread. Every thread can contain many scripts. Scripts contained in separate threads are executed concurrently. Scripts contained in the same thread are executed only when the other scripts belonging to the same thread are not executed. Every thread has its specific priority, which decides about the importance level of the scripts executed within it, i.e. script in

a thread with higher level can interrupt the operation of any script executed in thread with lower priority. Such an interruption usually occurs when script in thread with lower priority must response to an event. Such an event can be, for example, read operation of new process variable value.

Thread priority is defined with numerical value in the range 0 to 6. Priority values correspond with priorities of thread in Microsoft Windows and have the following meaning:

0 - idle
1 - lowest
2 - below normal
3 - normal - default
4 - above normal
5 - highest
6 - time critical

Priority 3 (normal) is default thread priority, i.e. thread has priority 3 if no other priority was set in the initialization file.

Script module creates threads upon initialization. Threads are removed when the Asix application is being completed. Every thread (except for default thread) has its name defined in the initialization file. Regardless of items in the initialization file, the script module always creates default thread. Default thread has no name and has priority 3 (normal). All the scripts, which are not assigned to a specific thread, are executed in default thread.

A script is assigned to a specific thread by entering appropriate execution parameter of the script. A parameter takes the following form:

//thread:*thread_name [:priority]*

where:
thread_name - arbitrarily selected thread name;
priority - number from the range 0- 6, which specifies thread priority.

The same thread name can occur in declarations of many scripts. All such scripts will be executed in the same thread. Priority of the same thread can be defined in one or a few declarations. If it occurs in a few declarations, it must have the same value.

On the basis of analysis of the initialization file, script module creates appropriate threads and assigns declared scripts to them.

EXAMPLE

```
reporter1 = reports.vbs //thread=reports:2
reporter2 = reports.vbs //thread=reports
calculators = calculator.vbs //thread=calculators:4
monitor = monitor.vbs
```

In the above-mentioned example, three threads will be created: default thread with priority 3 (normal), "reports" thread with priority 2 (below normal) and "calculators" thread with priority 4 (above normal). On the basis of the script program included in "reports.vbs" file, two scripts named "reporter1" and "reporter2" will be created. Both the scripts will be executed in the "reports"

thread. On the basis of the script program included in "calculator.vbs" file, a script named "calculators" will be created. The script will be executed in a thread named "calculators" with priority 4 (above normal). In "monitor" script declaration, the thread declaration is omitted. The script will be executed in default thread, which has priority 3 (normal).

Script priorities should be selected so that script execution does not disturb the operation of the Asix system. The main task of the Asix system, which functions as graphic user interface, is executed with priority 3 (normal). Scripts that operate on threads with the same or higher priority my cause that masks are refreshed more slowly if the tasks performed by the scripts requires too much processor time. Particular caution should be taken in case of scripts, which operate in threads with priorities higher than normal, as their operation can disturb functioning of other Asix components, such as measured data acquisition module and measured data archiving module.

Scripts belonging to the same thread are initialized in the order of the script declaration occurrence in the initialization file. Scripts are completed in reverse order. The initialization and completion order can be of significant importance for scripts that call functions from other scripts and exchanging data with other scripts.

Scripts that operate in various threads can exchange data between each other. It is also possible for a script to call functions, which are contained in another script and thread.

3.2 Control of Script Execution Time

Script's function execution time control enables to detect defective scripts and is performed by definition of the maximum script execution time (timeout).

Two time limits can be defined for every script: The first one is used during script initialization, while the later one is used to limit script operation time as a result of response to external event. Exceeding script operation time breaks it. Script will not be executed any more. Introduction of two different limits enables to determine the maximum execution time of the script during initialization, which may require long activities (such as activation of Excel or initialization of access to a database) and maximum time during normal script operation (response to events), which is usually shorter.

Maximum times for script execution are entered into application configuration file with use of Architect module:

Architect > *Fields and Computers* > *Scripts and Actions* module > *Scripts Parameters* tab

Event timeout – parameter determines the maximum time of event handling by the script. If the
parameter value is 0, the event handling time is not limited. The unit is time expressed in milliseconds.
Default value: 5000 milliseconds
Initialization timeout - parameter determines the maximum time of initialization part handling of the script. If the parameter value is 0, the initialization handling time is not limited. The unit is time expressed in milliseconds.
Default value: 15000 milliseconds

If the above-mentioned parameters are omitted, default value will be assumed.

The above-mentioned parameters determine limits for every script, which has not specific settings. For every script separate limit values can be defined. It is done with use of "**/ /IT:***nnn*" execution parameters for maximum initialization time and "**/ /I:***nnn*" for maximum time of response to an event. "*nnn*" determines time in milliseconds.

Script execution control time is disabled when the script is stopped as a result of step by step operation or setting breakpoint in script debugger.

3.3 Tracing Changes in Script File

The **asix**'s script module automatically traces changes of the write times of the file in which script program is saved. In case of a change in this time, which happens as a result of file editing or its replacing by another file, script module ends the operation of the currently executed script, reads out the script file

contents again and activates the script on the basis of a new program. The above-mentioned function enables to make corrections in the script during operation of the Asix system without necessity to stop the application.

3.4 Cooperation with Script Debugger

The Asix script module enables execution of scripts under control of script debugger. Operation under script debugger is possible only when script debugger is installed in the system. The below-described script module functions were tested with use of Microsoft Script Debugger. It is also possible to use debuggers included in Excel, Word, InterDev and FrontPage packages. Also, debuggers consistent with ActiveX ® Scripting technology may be used.

The main parameter responsible for cooperation is the DEBUGGER parameter in the application initialization file with use of Architect:

Architect > *Fields and Computers* > *Scripts and Actions* module > *Scripts Parameters* tab:

Debugger settings - parameter determines the scope of cooperation between scripting module and script debugger
-***Run scripts without debugger*** – if this option is selected, debugger will be never started; in
case of script timeout or error, AS program panel will be fed with error information with line
number and character given; executive parameters //X and //D given in the script declaration
are irrelevant, if option *Run scripts without debugger* is checked;
-***Run scripts with debugger –*** option allows to run scripts under control of the debugger;
-***Run scripts and start debugger at the moment when script module is initialized;***
Default value: *Run scripts without debugger*

Script's execution control time is disabled when the script is stopped as a result of step operation or setting breakpoint in script debugger.

4 General Form of Script

4.1 Script Components

Script usually consists of an initialization section and a section responding to an event. A script may consist of the initialization section only. It regards especially scripts initialized as a result of operator actions. Any activities in scripts of that type are performed during script activation. The other scripts perform initialization activities, including connecting to sources of events and then moves to the phase of waiting for an event.

Below, an example do a script with a separate initialization section and a section responding to an event.

Listing 1. An example of a script with a separate initialization section and a section responding to an event which includes acquisition of a new value of a process variable by the ASMEN module.

VBScript

```
‘ Initialization section
Set Panel = Asix.Panel
Panel.Message("Test OnRead")

Set Var = Asix.Variables("Emisja_CO2")

Var.OnRead = GetRef("OnRead")

‘ Section responding to an event
Function OnRead( val, stat, tim )
Panel.Message("New value")
End Function
```

JScript

```
// Initialization section
var Panel = Asix.Panel;
Panel.Message("Test OnRead");

var Var= Asix.Variables( "Emisja_CO2" );

Var.OnRead = OnRead;

// Section responding to an event
function OnRead( val, stat, time )
{
  Panel.Message("New value");
}
```

4.2 Script Deinitialization

Script has not a distinguished deinitialization section. In order to enable programming of the actions performed during completion of operation by the script, such as release of occupied resources, the *OnTerminate* event of an *Asix* object and the *OnTerminate* event of a *Script* object were introduced.

Listing 2 . An examplary script with the 'OnTerminate' event of an Asix object and the 'OnTerminate' event of a Script object.

VBScript

```
‘ Initialization section
Asix.OnTerminate=GetRef("OnASIXTerminate")
Asix.Script.OnTerminate   = GetRef("OnTerminate")
‘ Section responding to an event
Function OnASIXTerminate()
‘...
End Function

Function OnTerminate(a)
‘...
End Function
```

JScript

```
// Initialization section
Asix.OnTerminate = OnASIXTerminate;
Asix.Script.OnTerminate = OnTerminate;
// Section responding to an event
function OnASIXTerminate()
{
  //...
}
function OnTerminate()
{
  //...
}
```

The *OnASIXTerminate* and *OnTerminate* events are executed during closing the Asix application. They are executed only when the script is active during closing application (it has been activated by ***Scripts*** parameter). *OnASIXTerminate* is executed before the *OnTerminate* event.

4.3 Access to asix Resources

Access to Asix resources is performed with use of a set of objects made available by the script module. Every script activated by script module has direct access to *Asix* – the main object made available by the script module. Access to other objects enabled by *Asix* itself and other objects acquired by *Asix*.

Listing 3. An exemplary script with an access to the Panel object and an output of the "message" message by this object.

VBScript

```
Dim Panel
Set Panel = Asix.Panel
Panel.Message( "message")
```

JScript

```
var Panel = Asix.Panel;
Panel.Message( "message");
```

The above-mentioned example shows the way of an access to *Panel* and an output of the "message" message by means of it. It is not necessary to assign the object (here *Panel*) to the language variable (in this case it is the "Panel" variable), but it can affect the script execution time if there are frequent references to a given object. If a script outputs messages frequently, it is more advantageous to store *Panel* in a separate variable.

Listing 4. An exemplary script with the output of a message without the use of an additional variable.

VBScript
```
Asix.Panel.Message( "message")
```

JScript
```
Asix.Panel.Message( "message");
```

The above is an example of the output of a message without the use of an additional variable.

One of the most important objects made available by the script module is *Variable*, which represents the process variable. The *Variable* objects returned by *Asix* as a result of execution of the *Variables* function.

Listing 5. An exemplary script realizing an access to an object that represents the variable named "Emission_CO2".

VBScript
```
Set Var = Asix.Variables("Emission_CO2")
```

JScript
```
var Var= Asix.Variables( "Emission_CO2" );
```

The above example shows the access to an object that represents the variable named "Emission_CO2".

4.4 Response to Events

You can assign a specific function to an event with the use of the following expression:

Listing 6. Assigning a specific function to an event.

VBScript
```
object.event_name= GetRef("function_name")
```

JScript
```
object.event_name = function_name;
```

Where "object" is an expression returning a specific object (e.g. variable name, which relates to a specific object), "event_name" is the name of an event generated by the "object" object and the "function_name" is the function name (or **Sub** sub-program in case of VBScript), which will be executed upon occurrence of an event.

It should be stressed here that in order to execute a function of response to an event, a specific object must exist when the event appears.

Listing 7. An exemplary script with erroneous assigning a function to an event.

VBScript

```
F()
Function F()
Dim Var
Set Var = Asix.Variables("Emi_CO2")
Var.OnRead = GetRef("OnRead")
End Function

Function OnRead( val, stat, tim )
End Function
```

JScript

```
F();
function F()
{
var Var= Asix.Variables( " Emi_CO2" );
Var.OnRead = OnRead;
}

function OnRead( val, stat, time )
{
}
```

The above is an example of erroneous assigning a function to an event. After the "F" function has been completed, the "Var" variable, which stores a *Variable*-type object, as a local variable of the "F" function ceases to exist together with the *Variable*-type object. In order make an assignation of a function to an event efficient, the generating object must be stored in another place (e.g. in global variable).

The name of the function of response to an event in the above-mentioned examples was the same as the event's name. In general case, function name is not related to the event name and it can be any name.

In the function of response to an event there is an object, which has generated the event. The object, which generates the event, is accessible by means of **this** in JScript language and **Me** in VBScript.

Listing 8. An exemplary script with an output of a variable name to the asix control panel by means of the OnRead function.

VBScript

```
Dim Var
Set Var = Asix.Variables("Emi_CO2")
Var.OnRead = GetRef("OnRead")
Function OnRead( val, stat, tim )
call Asix.Panel.Message(Me.Name)
End Function
```

JScript

```
var Var= Asix.Variables( "Emi_CO2" );
Var.OnRead = OnRead;

function OnRead( val, stat, time )
{
Asix.Panel.Message(this.Name);
}
```

In the above-mentioned example, the "OnRead" function will output the name of variable, which measured value has been read out, to control panel of the Asix system.

In order to remove connection between the event source and function of response to an event, the empty value should be given as a function of response to an event.

Listing 9. The way of passing the empty value as a function of response to an event.

VBScript	JScript
Var.OnRead = **Empty**	Var.OnRead = **null**;

The functions of event handling are executed after the execution of the current fragment of the script has been completed, i.e. the initialization section of the script or the function of event handling. It means that the current script instruction sequence is not interrupted by the function of event handling. The exception to this rule can be calling a method of an external object, i.e. the object from outside the set of objects made available by script modules. The current version of the script module does not include the methods admitting the execution of the event handling function during execution of the method.

5 Function of Object Model of Script Module

5.1 Access to Measured Data

The *Variables* method of an *Asix* object provides access to Asix process variables. As a result of its execution, the value is returned, which is an object representing the variable. The argument is the archive name and type. In case of failure, for example when a given data is missing in the ASMEN database, the method returns the empty value, which may be used in the script to detect erroneous variable names.

Listing 10. An exemplary script that realizes an access to an asix process variable along with a detection of erroneous variable names.

VBScript

```
Set Var = Asix.Variables("Emi_CO2")
If TypeName(Var) = "Nothing" Then
Asix.Panel.Message("Wrong name" )
End If
```

JScript

```
var Var= Asix.Variables( "Emi_CO2" );
if ( Var == null )
Asix.Panel.Message("Wrong name" );
```

Each time a function with the same argument is called, a new object referring to the same Asix variable is returned. Lifetime of such an object is the time of existence of reference to this object in the script.

The *Variable* object includes the *Value*, *Status* and *Time* components, which define the value, status and time of the last read measured data. Reading the measured data is executed automatically by the ASMEN module of the Asix system in the refreshing cycle resulting from definition of the process variable. In order to read the current measured value, the *Read* method of a *Variable* object can also be used – it forces to refresh the process variable value upon execution of the method. The values of the *Value*, *Status* and *Time* fields are not changed during execution of the current fragment of the script, i.e. the initialization section of the script or the event handling function. The exception to this rule can be calling an external object method, i.e. the object from outside the set of objects made available by script modules or the script module methods, which purpose is to change the value of the fields (e.g.: *Read*, *ArcRead*, record to the *Current* field, etc.).

5.2 Access to Historical Data

If the argument of the *Variables* function of an *Asix* object is the variable name with a single-letter code of archive type, the script module will try to open the archival measurement database. If the trial will be a success, the logical value of the *WithArchive* field of an *Info* object will be **true** (access to an *Info* object takes place through the *Info* field of a *Variable* object). The *ArcRead*, *ArcReadAt*, *ArcSeek* and *ArcTell* methods can be then used to review the archive data. One should pay attention that the archival data values are returned by a *Variable* object by means of *Value*, *Status* and *Time* components, which are used also to return the current values. In order to avoid ambiguousness, a *Variable* object has the *Current* field, which value is **true** if the current values are returned or **false** in the opposite case. Execution of any function related to access to archive data, sets the field to **false**. If the *Current* field's value is **false**, the *OnRead* event callback function is not called. In order to return access to the current measurements, the logical value of the *Current* field should be set to **true**. It will update the *Value*, *Status* and *Time* fields with the current values.

Below, an example of script that retrieves value of a variable named "Emi" from the D-type archive is presented. The value of January 20th, 2002 at 17:50:31 is being retrieved. The script example contains calling of the *ArcParam* method of a *Variable* object with the INTR_NEAREST parameter. The call of this method will return the closest value if the archive does not contain any measured data for the specified moment. If operation is successful, the retrieved value is entered into the control panel.

Listing 11. An exemplary script reading the variable named "Emi" form the D-type archive.

VBScript

```
Dim Stat
Dim Panel
Dim Var
Set Panel = Asix.Panel

Set Var = Asix.Variables("Emi,D")

' Determination of data interpolation method – interpolation to the closest data.
Var.ArcParam( INTR_NEAREST )

Stat = Var.ArcReadAt(#1/20/2002 17:50:31#)

If Stat = STAT_OK Then
If Var.Status = OPC_QUALITY_GOOD Then
Panel.Message("Emi= " & _
CStr(Var.Value) & " mg/h" )
End If
End If
```

JScript

```
var Stat;
var Panel = Asix.Panel;
var Var = Asix.Variables("Emi,D");
var Time = new Date(2002, 0, 20, 17, 50, 31);

// Determination of data interpolation method -
// interpolation to the closest data.
'Var.ArcParam( INTR_NEAREST );

Stat = Var.ArcReadAt(Time.getVarDate());

if ( Stat == STAT_OK )
if ( Var.Status == OPC_QUALITY_GOOD )
Panel.Message("Emisja= " + Var.Value + " mg/h"
);
```

5.3 Data Status

Measured data status is expressed as that of OPC (OLE for Process Control). It regards the *Status* field of the *Variable* variable, the *OnRead* event handling function, the *Write* function of a *Variable* object and others. Using OPC statuses, symbolic values can be used (see the table below).

Table 1. The values of measured data statuses.

Symbolic value	**Numerical value (hexadecimal code)**
OPC_QUALITY_MASK	C0
OPC_QUALITY_BAD	00
OPC_QUALITY_CONFIG_ERROR	04
OPC_QUALITY_NOT_CONNECTED	08
OPC_QUALITY_DEVICE_FAILURE	0c
OPC_QUALITY_SENSOR_FAILURE	10
OPC_QUALITY_LAST_KNOWN	14
OPC_QUALITY_COMM_FAILURE	18
OPC_QUALITY_OUT_OF_SERVICE	1C
OPC_QUALITY_UNCERTAIN	40
OPC_QUALITY_SENSOR_CAL	50
OPC_QUALITY_EGU_EXCEEDED	54
OPC_QUALITY_SUB_NORMAL	58
OPC_QUALITY_LAST_USABLE	44
OPC_QUALITY_GOOD	C0
OPC_QUALITY_LOCAL_OVERRIDE	D8
OPC_LIMIT_MASK	03
OPC_LIMIT_OK	00
OPC_LIMIT_LOW	01
OPC_LIMIT_HIGH	02
OPC_LIMIT_CONST	03

The use of OPC statuses requires an appropriate configuration of the Asix ASMEN module. The following record should be included in the application configuration file with use of Architect:

Architect > *Fields and Computers* > *Current data* module > *Advanced* tab > ***OPC status*** parameter

If ASMEN does not use OPC status standards, the script module will output a warning message to the Asix control panel. A *Variable* object cannot be created then.

In addition to the values defined in OPC standard, in Asix statuses specific to this system are defined. They take the measured data status bits 8-15 (see the table below).

Table 2. The data statuses typical of the Asix system.

Symbolic value	Numerical value (hexadecimal code)	Description
AS_STAT_MASK_SOURCE	0300	Bit mask of the field, which defines the data source.
AS_STAT_NORMAL	0000	Data acquired in the basic mode.
AS_STAT_HIST	0100	Historical data.
AS_STAT_MANU	0200	Data entered manually.
AS_STAT_SIMUL	0300	Simulated data.
AS_STAT_MASK_ORIGIN	0400	Bit mask of the field, which defines the data source (ASPAD).
AS_STAT_LOCAL	0000	Locally acquired data.
AS_STAT_COPY	0400	Data copied from different archive.
AS_STAT_SECOND_LIMIT	0800	Together with OPC_LIMIT_HIGH or OPC_LIMIT_LOW means HIGHHIGH or LOWLOW limit has been exceeded.
AS_STAT_UTC_TIME	4000	Data time is expressed as UTC. (see: Remarks Regarding Time.)
AS_STAT_MASK_OPC_STAT	00FF	Bit mask, which determined the youngest status byte, where 8-bit status according to OPC standard is contained.

5.4 Dynamic Object Fields

Majority of the objects of the script module permit to add new fields. It is allowed by linking information with an object they relate to. It also allows interrelating some objects. The example of the application of dynamic fields and interrelation of objects may be a job consisting in calculation of each value from specific set of variables (source variables) and recording the result in another variable (destination variables). Every source variable is related to a specific destination variable, while the values are counted and the results are recorded in the event response function *OnRead*. The *OnRead* event is generated by a *Variable* object after the ASMEN module acquires new measured values.

Listing 12. An exemplary script including dynamic components of objects.

VBScript

```
Set VarSrc1 = Asix.Variables("Src1")
VarSrc1.Set(„Dst") = Asix.Variables("Dst1")
VarSrc1.OnRead = GetRef("OnRead")

Set VarSrc2 = Asix.Variables("Src2")
VarSrc2.Set(„Dst") = Asix.Variables("Dst2")
VarSrc2.OnRead = GetRef("OnRead")
‘e.t.c.
Function OnRead( val, stat, tim )
call Me.Dst.Write( -val, stat, time )
End Function
```

JScript

```
var VarSrc1= Asix.Variables( "Src1" );
VarSrc1.Dst = Asix.Variables( "Dst1" );
VarSrc1.OnRead = OnRead;

var VarSrc2= Asix.Variables( "Src2" );
VarSrc2.Dst = Asix.Variables( "Dst2" );
VarSrc2.OnRead = OnRead;
//e.t.c.
function OnRead( val, stat, time )
{
this.Dst.Write( -val, stat, time );
}
```

In the above example, the source variable name type is "Src*n*" and the destination variable name type is "Dst*n*". Every object representing a source variable has a dynamically added field named "Dst". This field will store a reference to the object representing a destination variable. In the *OnRead* event callback function (common for source variables), the "Dst" dynamic field is used to locate relevant destination variable and record a value of source variable into it. The example also presents how to use **Me** (VBScript) and **this** (JScript) keywords. The keywords enable access to the object for which the event response function is being executed and this, in turn, enables to write one comprehensive event response function for many variables. Of course, in case of higher number of variables, instead of a series of "**var** VarSrc*n*" declarations, the source variables should be rather placed in any structure (e.g. table), which gathers all the source variables. For destination variables no structure is required as they are stored in "Dst" dynamic fields of source variables.

REMARK The objects, which permit to create dynamic fields can reserve a certain set of field names to be used in the future. The reserved field names must

not be used in the script. Every object reserves the names of *Diagnostics* and *Explicite*.

REMARK It should be noticed here that VBScript is not case-sensitive during access to the fields of the object. If two dynamic fields named "aa" i "AA" are created in Script:

object.aa = 5;
object.AA = 6;

in VBScript access to the "aa" field will only be possible, regardless of which object construction is used - "object.aa" or "object.AA".

In JScript, dynamic fields are created with use of field access operator ('**.**') in the assigning instruction (as in the above-mentioned example). If the field does not exist, a new value will be created and assigned to it. In VBScript, the same method cannot be used. The instruction of assigning to non-existing field ends with error. For this purpose, every object, which enables to add filelds dynamically, makes the *Set* method available, the call of which has the following form:

object.Set(*name*) = *value*

where: *name* is a string of characters between quotation marks that defines the name of new field.

If during the execution of the *Set* method, the field with the name specified does not exist, it will be created.

Similarly to the *Set* method, every object that permits to add dynamic fields has the *Get* method, which returns the field value.

object.Get(*name*)

The use of the *Get* method is usually not necessary. The exception is the operation of taking the dynamic object method in VBScript, i.e. when the method should be treated as a field and acquires its contents as the reference to the method.

```
Asix.GlobalData.Set("Calculator") = GetRef("Calculator") 'dynamic method
Set Calculator = Asix.GlobalData.Get("Calculator") 'assigning the contents
of the field named Calculator
Calculator() 'function call
```

Due to VBScript syntax, the instruction:

```
Set Calculator = Asix.GlobalData.Calculator
```

does not allow unambiguous interpretation as the field getting. VBScript allows to omit the function brackets when it has no parameters then the above instruction can be interpreted as a method call. In such a situation, the *Get* method allows to avoid ambiguousness.

Both the methods can be used in any language. These methods can also be used to create and access the fields, which names conflict with rules of the script language.

The value of the object's dynamic field may be not only a primitive data (number, text), but also an object and method (function).

In the case when the value of dynamic field is a function, it can be used in the same way as the method of the object. In particular, if the keywords **Me** in VBScript and **this** in JScript are used inside such a function, they relate to the object, the function is a field of.

EXAMPLE

Listing 13. An exemplary script including the dynamic component in the form of 'Fun' named method added to a 'Variable' object.

VBScript

```
Set Var = Asix.Variables("Emi_CO2")
Var.Set("Fun") = GetRef("Fun")
Var.Fun()
Function Fun()
Asix.Panel.Message( "Var " + Me.Name )
End Function
```

JScript

```
var Var= Asix.Variables( "Emi_CO2" );
Var.Fun = Fun;
Var.Fun(); //call of dynamic method
Function Fun()
{
Asix.Panel.Message( "Var " + this.Name );
}
```

In the above example, a dynamic component in the form of a method named "Fun" was added to a *Variable* object. Execution of both the scripts will cause that a message "Var Emi_CO2" will be output to the Asix control panel. Call of this function as an independent function ("Fun()" instruction and not "Var.Fun()") will end with error because in such a case the keywords **Me** and **this** do not relate to the *Variable* object.

REMARK It should be avoided to create interrelations between the objects that form a closed cycle.

5.5 Calculation and Recording Measured Data

Scripts can be used to develop calculating functions, which carry out non-standard calculations of the values of measured data. With the use of script, however, it is not possible to modify the value retrieved from external device as the ASMEN module does by means of calculating functions. However, it is possible to use the *Write* function of the *Variable* object to record the result of calculations of the values of one variable to another one, which is declared in the NONE channel and functions as a buffer of counted values only. The result of calculations should be usually related to time and status. In order to assign a new value of the variable of the NONE channel together with the status, the NONE channel must be configured in appropriate way. The ***Writing data and status*** parameter should be declared for the NONE driver channel.

If this parameter is missing, the variable value will be recorded only.

The *Write* method of the *Variable* object has the following form:

Write(*value* [, *status* [, *time*]])

If *status* parameter is omitted, it is assumed to be equal to OPC_QUALITY_GOOD i.e. the data is correct, while if *time* parameter is omitted the current time is assumed. If status and time of a given measured data are unimportant, the *Value* field can be used to assign a new value to the variable, i.e. instructions:

object.Write(5)
and
object.Value = 5

are equivalent. (*object* means a *Variable* object)

If the NONE channel has not the ***Writing data and status*** parameter, the status and time given during the recording operation are ignored. In case of physical transmission channels, the ability of time and status recording depends on a specific channel.

Recording to a variable can also be used to create simulated trends and to record a value to external devices.

Additional information on execution of the counting functions can be found in item "Creating FunctionLibraries" and "Remarks Regarding Time".

NOTE Recording to a variable by means of assign operation to *Value* field or *Write* method, causes that the new value is physically transferred to the external device (or NONE channel). It does not mean the *Value*, *Status* and *Time* fields are updated automatically. The physical reading from external device can update the fields only. After the recording has been completed, the Asix ASMEN module automatically reads physically new values, which causes the above-mentioned

fields are updated and OnRead event is generated. However, the fields will be updated only after the current event or the initialization section of the script has been completed. In order to force updating the fields, the Read method should be performed.

5.6 Array Variables

The ASMEN module of the Asix system enables to read and write array variables. An array variable is a set of measuring data, which consists of many data of the same type. One status and one time are related to all the data included in an array variable. If a *Variable* object relates to an array variable, the *Value* field and the "*value*" parameter of the *OnRead* event callback function is an array as well. It is an array in VBScript meaning. A data written into the *Value* field and the "*value*" parameter of the *Write* method must also be an array of this type. Additionally, in JScript, this data can be an *Array* object of JScript. The data written should include at least as many components as the variable concerned by the *Variable* object includes. The number of components, an array variable consists of, can be read from the *Count* field in an *Info* object. Attention should be paid to the fact that in case of array variables the way of access to specific values of the variable changes. A number of specific data in the array, i.e. index, should be used. The number of the first element in the array is 0. JScript does not allow reading individual values of the array, which is in the *Value* field or the "*value*" parameter in the *OnRead* function. Such array should be converted into a *VBArray* object and only then the individual values forming array variable can be read from this object. However, the content of a *VBArray* object cannot be modified. Modification of *VBArray* object table elements is possible after the object is converted into an *Array* object with the *toArray* method of a *VBArray* object.

Below, a few examples of handling array variables in scripts are presented.

EXAMPLE 1

Listing 14. Conversion of values from one array variable into another.

VBScript

```
Set Var1 = Asix.Variables("Emi_CO")
Set Var2 = Asix.Variables("Emi_CO2")

Var1.Value = Var2.Value
‘or
Var1.Write(Var2.Value) ’status and time can be given
```

Jscript

```
var Var1= Asix.Variables( "Emi_CO" );
var Var2= Asix.Variables( "Emi_CO2" );

Var1.Value = Var2.Value;
//or
Var1.Write(Var2.Value); //status and time can be given
```

In the above-mentioned example, in JScript no conversion of the table into intermediate objects was necessary as the *Write* method and the *Value* field accept tables in the meaning of VBScript.

EXAMPLE 2

Listing 15. Readout of two-element table from one variable, modification of its elements and writing into another variable.

VBScript

```
Set Var1 = Asix.Variables("Emi_CO")
Set Var2 = Asix.Variables("Emi_CO2")

Dim  arr

arr = Var1.Value

arr(0) = cint(arr(0)) +1
arr(1) = cint(arr(1)) +1

Var2.Value = arr
‘or
Var1.Write(Var2.Value) ’status and time can be given
```

JScript

```
var Var1= Asix.Variables( "Emi_CO" );
var Var2= Asix.Variables( "Emi_CO2" );
//conversion to VBArray
var arr1 = new VBArray(Var1.Value);
//conversion to Array
var arr2 = arr1.toArray();

arr2[0] += 1;
arr2[1] += 1;

Var2.Value = arr2;
//or
Var2.Write(arr2);  //status and time can be given
```

In the above-mentioned example, to every array element of the "Var1" variable 1 was added. The resulting array was saved into "Var2". In case of JScript, it was necessary to convert into *VBArray*, and then into *Array*.

EXAMPLE 3

Listing 16. Copying of new values into two-element array variable.

VBScript

```
Set Var = Asix.Variables("Emi_CO")

Dim arr(1)
arr(0) = 1
arr(1) = 2

Var.Value = arr
‘or
Var.Write(arr)  ’status and time can be given
’Writing with use of Array function
Set Var = Asix.Variables("Emi_CO")

Var.Value = Array(1,2)
‘or
Var.Write(Array(1,2)) ’status and time can be given
```

Jscript

```
var Var= Asix.Variables( "Emi_CO" );

var arr = new Array(1,2);

Var.Value = arr;
//or
Var.Write(arr); //status and time can be given
```

EXAMPLE 4

Listing 17. Access to individual array elements in the event handling function OnRead.

VBScript

```
Set Var = Asix.Variables("Emi_CO")

Var.OnRead = GetRef("OnRead")

Function OnRead( val, stat, tim )
Asix.Panel.Message "Elements:" & CStr(val(0)) &
CStr(val(1))
End Function
```

JScript

```
var Var= Asix.Variables( "Emi_CO" );

Var.OnRead = OnRead;

function OnRead( val, stat, time )
{
  var arr = new VBArray(val);
  Panel.Message("Elements:"      +arr.getItem(0)
+"," +arr.getItem(1);
}
```

The *OnRead* function outputs a message containing values of individual array elements into the Asix control panel. In JScript, conversion of the "val" parameter into the *VBArray* object was necessary.

5.7 Array Variables and Operations on Texts

The SCRIPTS module allows to assign the text to an array variable the elements of which have the length of one byte. These variables are usually associated with the NOTHING_TEXT conversion function. Assigning the text consists in passing the consecutive text characters to the consecutive elements of an array variable. The text may be of any length. If the number of text characters is less than the number of array variable elements, the residual elements of the variable take the value of 0. If the number of text characters is larger than the number of array variable elements, the residual elements of the variable are ignored.

EXAMPLE

Listing 18. An exemplary script presenting the way of operating on texts.

VBScript

```
Set Var = Asix.Variables("Text")
Var.Value = "This is a text"
Set Dzisiaj = Now()
Var.Value = " Today is " & Today
```

JScript

```
var Var= Asix.Variables( "Text" );
Var.Value = " This is a text";
var Dzisiaj = new Date();
Var.Value = "Today is " + Today;
Var.Value = new String("This is a text");
```

The way of text operation described below is handled by the SCRIPTS module in v 1.06.000 or upper.

5.8 Access to Alarm System

The object model of the script module enables to generate and check the status of Asix alarms. Alarms are identified by means of numbers and are declared in appropriate configuration files. For information on this subject, see the Asix system documentation.

The *Raise*, *Cancel* and *State* methods in an *Asix* object are used to set, reset and check the alarm status. The *Raise* and *Cancel* function parameters enable to set two alarm parameters and time (see: Remarks Regarding Time).

5.9 Execution of Operator Action

An operator action can be executed by means of the *ExecuteAction* method. A parameter of this method is a text that specifies the action to be executed. For detailed information on operator actions see specification of Asix.

EXAMPLE

```
Asix.ExecuteAction( "Open Emi.msk" )
```

The above command will open "Emi.msk" mask.

5.10 Access to Contents of Initialization File

The object model enables to access the items in the initialization file sections. It is possible due to the *Parameters* method in an *Asix* object. This method returns object-collection of all the parameters in the section with a name defined in the *Parameters* function parameter. Every element of this collection is an object containing the *Name* and *Value* fields, which return the parameter name and its value. As *Value* is a default field, its name can be omitted. The object returned by the *Parameters* function as collection has the *Count* field, which determines number of parameters in a section (or the number of collection elements) and the *Item* method, which enables to access the parameter by entering its number (beginning from 1). Parameter names in the initialization file section can repeat.

Listing 19. An example of viewing the section named xxx of the INI file.

VBScript

```
Set Parameters = Asix.Parameters("xxx")
For Each Parameter in Parameters
Dim keyName
keyName  = UCase(Parameter.Name)
Select Case keyName
Case "VARIABLE"
‘.......
Case "DIAGNOSTICS"
‘.....
Case Else
Call Asix.Panel.Error( "Wrong parameter " + _
Parameter.Name + "=" + Parameter)
End Select
Next
```

JScript

```
var Parameters = Asix.Parameters("xxx");
for ( Parameter in Parameters )
{
var Name = new
String(Parameters[Parameter].Name);
Name = Name.toUpperCase();
switch (Name.valueOf() )
{
case "VARIABLE":
break;
case "DIAGNOSTICS":
break;
default:
Panel.Error( "Wrong parameter " +
Parameters[Parameter].Name + "=" +
Parameters[Parameter]);
break;
}
}
```

EXAMPLE

Listing 20. An example of access to the third parameter by its number.

VBScript

```
Set Parameters = Asix.Parameters("xxx")
If Parametrs.Count >= 3 Then
Set xx = Parameters (3) ‘third parameter
End If
```

JScript

```
var Parameters = Asix.Parameters("xxx");
if ( Parameters.Count >= 3 )
xx = Parameters.Item(3) //third parameter
```

5.11 Access to Parameters Located in Script Declaration

The parameters passed to a script can be located in the script declaration in the initialization file and in the contents of the SCRIPT action.

Inside the script the parameters can be read with the use of the *Arguments* field in a *Script* object (a *Script* object is the value of the *Script* field in an *Asix* object). The value of the *Argument* field is an object-parameter collection. This object as a collection has the *Count* field and the *Item* method.

Listing 21. An exemplary script which outputs parameter values into control panel.

VBScript

```
Set Parameters = Asix.Script.Arguments

For each Parameter in Parameters
Asix.Panel.Message( "Parameter " +
Parameter)
```

JScript

```
Parameters = Asix.Script.Arguments;

for ( Parameter in Parameters )
Asix.Panel.Message( "Parameter=" +
Parameters[Parameter]);
```

Parameters are also accessible by using parameter number (the first is 1).

Listing 22. A script declaring a parameter number.

VBScript

```
Xx = Asix.Script.Arguments (1) ‘first parameter
```

JScript

```
xx = Asix.Script.Arguments.Item(1) //first
parameter
```

Collection contained in the *Asix.Script.Arguments* field has the same fields as the collection of parameters in the initialization file section described in previous item.

5.12 Events Related to Time

- SetInterval method
- ExecuteAt method

The *SetInterval* and *ExecuteAt* functions in an *Asix* object enable to execute a function at a given time. The *SetInterval* method makes cyclic call of the function defined by this parameter, and the *ExecuteAt* method makes single execution of the function at a given time. The functions return the pair of function identifier and a time assigned to it. This pair can be used for canceling such an assignment. In addition, the *SetInterval* method has a parameter, which permits to determine time phase in relation to interval multiple.

The ***SetInterval*** method has the following form:

SetInterval(*function, interval*[, *phase*])

Time parameters are expressed in milliseconds.

Call in the form of *SetInterval(function,5000)* will make cyclic calling of functions every five seconds, while *SetInterval(function,5000,0)* call will make the function be called at 00:00:00, 00:00:05, 00:00:10, etc. Call in the form of *SetInterval(function,5000,1000)* will make the function be called at o 00:00:01, 00:00:06, 00:00:11, etc. If *Interval* is a negative number, the *function* will be executed only once.

The ***ExecuteAt*** methodhas the following form:

ExecuteAt(*function, time*)

Time is determined as the total time (see: "Remarks Regarding Time").

EXAMPLE

Listing 23. An example of cyclic function call.

VBScript

```
Call Asix.SetInterval( GetRef("Timer"), 1000 )

'Function is executed every 1 second
Function Timer()
'.....
End Function
```

JScript

```
Asix.SetInterval( Timer, 1000 );

//Function is executed every 1 second
function Timer()
{
//.....
}
```

Linking the function with time is removed with the *ClearInterval* function.

5.13 Access to External Resources

Scripts executed by the script module can have access to external resources, such as spreadsheets, e-mail, etc. Access to external resources takes place by means of objects created by functions built into script language. Below, start-up of Excel is presented.

Listing 24. An exemplary script with Excel start-up.

VBScript

```
Dim Excel
Set Excel = CreateObject("Excel.Application")
```

JScript

```
var Excel = new ActiveXObject("Excel.Application");
```

In addition to access to external objects that are not connected with scripts directly, there are objects, which main purpose is to extend the range of functions performed by the script. The basic example of such objects is a *FileSystemObject*, which makes possible the access to files and directories. The *FileSystemObject* is created in the following way.

Listing 25. An exemplary script creating a 'FileSystemObject'.

VBScript

```
Dim fso
Set fso = CreateObject("Scripting.FileSystemObject")
```

JScript

```
var fso = new ActiveXObject("Scripting.FileSystemObject ");
```

Another example is the objects accessed by Windows Script Host, which access to network resources, system registry, program start-ups and other functions. Below, some objects are listed:

WshShell	allows creating processes and shortcuts. Provides access to the operating system registry. Allows recording the information into log file.
WshEnvironment	provides access to environmental variables.
WshSpecialFolders	provides access to directories such as "Desktop", "My documents", etc.

Below, an example of access to the system's environmental variables is presented:

Listing 26. An exemplary script with an output of message about the number of computer system processors.

VBScript

```
Set WshShell = CreateObject("WScript.Shell")
```

JScript

```
WshShell = new ActiveXObject("WScript.Shell");
```

```
Set Env = WshShell.Environment("SYSTEM")
Asix.Panel.Message( "Number of processors="_
 &Env("NUMBER_OF_PROCESSORS"))
```

```
Env = WshShell.Environment("SYSTEM");
Asix.Panel.Message( "Number of processors=" +
Env("NUMBER_OF_PROCESSORS"));
```

Both the scripts output the message about the computer system processors number.

5.14 Data Exchange Between Scripts

The *Asix* object has two fields: *GlobalData* and *GlobalThreadData*, which are designed to exchange data between scripts. Both the fields return objects, which have only the fields assigned to them by scripts. The fields of a *GlobalData* object are available to all the scripts handled by the script module. As the time of access to the object is long, a *ThreadGlobalData* object was introduced. Every thread created by the script module has one own *ThreadGlobalData* object. By means of the object data can be exchanged between the scripts belonging to the same thread. The time of access to a *GlobalData* object is approx. 30 times longer than the time of access to a *GlobalThreadData* object.

A new component of the global object is created by creation of a new dynamic component in a way described in section "Dynamic Object Fields". The data can be exchanged between scripts written in different languages.

5.15 Creating Function Libraries

As both fields and methods can be dynamically added, this can be used to create function libraries used by other scripts. It also enables to use functions written in other programming languages. In order to create a library, the library functions should be placed in a separate file and assigned to the *GlobalData* or *ThreadGlobalData* global object. For example, a script named "Sum.js" can contain a function named "Sum", which sums up arguments and returns the result. The "Sum" function can be then used in VBScript.

Listing 27. An exemplary script "Sum.js" with the function that sums up arguments and returns the result.

```
Asix.GlobalData.Sum = Sum;  //"GlobalData" fields are available for every script

function Sum(a,b)
```

```
{
  return a+b;
}
```

The above-mentioned function can be used in a script included in "Calc.vbs" file in the VBScript language:

```
Asix.Panel.Message( "Sum=" + CStr(Asix.GlobalData.Sum(5,2)))
```

When "Calc.vbs" is run, a message "Sum=7" will be output to the **asix**'s control panel.

In order for the above-mentioned example to function properly, the initialization section of the "Sum.js" script must be executed before the initialization section of the "Calc.vbs" script. To this end, script declarations in the ***Scripts*** parameter of the initialization application file must be put in correct order (with use of Architect program):

```
Sum = Sum.js
Calc = Calc.vbs
```

The above-mentioned example will function properly only when both the library script and the script using the library script are in the same thread. Only then the order of script declarations corresponds with the order of script initialization. If library script were put to different thread, its initialization would be concurrent to initialization of the script that uses library functions, which in turn could result in an error in access to non-initialized components of a *GlobalData* object.

If library script and usable script are in the same thread, *ThreadGlobalData* should be used rather than *GlobalData*. The time of access to *GlobalData* is approx. 30 times longer than time of access to *GlobalThreadData*.

Taking into consideration the fact that by use of **Me** and **this** keywords an object it is a component of is available inside the method, the methods that will function in the context of *Variable* can be placed in the library module.

Listing 28. An exemplary script with a library module.

VBScript

```
Asix.GlobalData.Set("Calculator") = _
                        GetRef(„Calculator")
Function Calculator()
‘access to variable value using Me
End Function
```

JScript

```
Asix.GlobalData.Calculator = Calculator;
function Calculator()
{
// access to variable value using this
}
```

Listing 29. An example of a script of the module that uses a library.

VBScript

```
Set Var = Asix.Variables("Emi_CO2")
Var.Set("Calculator") = GlobalData.Calculator
Var.OnRead = GetRef(„OnRead")
Function OnRead()
Me.Calculator( ) ‘library call
End Function
```

JScript

```
var Var= Asix.Variables( "Emi_CO2" );
Var.Caculator = GlobalData.Calculator;
Var.OnRead= OnRead; //call of dynamic method
function OnRead()
{
this.Calculator(); //library call
}
```

The same can be obtained if the "Calculator" function has a parameter related to the *Variable* object.

Listing 30. An exemplary script with a library module (when the ‘Calculator’ function has the parameter relating to the Variable object).

VBScript

```
Asix.GlobalData.Set("Calculator") = _
                        GetRef("Calculator")
Function Calculator( Var )
  ‘access to variable value by Var parameter
End Function
```

JScript

```
Asix.GlobalData.Calculator = Calculator;
function Calculator( Var )
{
  //access to variable value by Var parameter
}
```

Module that uses library:

Listing 31. An example of a script of the module that uses a library (when the ‘Calculator’ function has the parameter relating to the Variable object).

VBScript

```
Set Var = Asix.Variables("Emi_CO2")
Var.OnRead = GetRef(„OnRead")
Function OnRead(val, stat, time)
‘ Library call
call Asix.GlobalData.Calculator(Me)
End Function
```

JScript

```
var Var= Asix.Variables( "Emi_CO2" );
Var.OnRead= OnRead; //call of dynamic method
function OnRead(val, stat, time)
{
// Library call
Asix.GlobalData.Calculator(this);
}
```

In the above-mentioned example, the solution used is disadvantageous due to time of execution. It consists in calling to the *GlobalData* object inside the event handling function every time. More efficient solution involves saving the "Calculator" function in a script's local variable in order to call it later.

Listing 32. An example of a script of the module that uses a library (the ‘Calculator’ function is stored in the local variable in order to a later call).

VBScript

```
Set Var = Asix.Variables("Emi_CO2")
Set                    Calculator
Asix.GlobalData.Get("Calculator")
Var.OnRead = GetRef(„OnRead")
Function OnRead(val, stat, time)
call Calculator(Me) ‘library call
End Function
```

JScript

```
 var Var= Asix.Variables( "Emi_CO2" );
=var Calculator= Asix.GlobalData.Calculator;
 //call of dynamic method
 Var.OnRead= OnRead;
 function OnRead(val, stat, time)
 {
 Calculator(this); //library call
 }
```

Of course, if the "Calculator" function performs a simple calculation of the measured value and does not require access to other fields of a *Variable* object, instead of sending the *Variable* object it is possible to send the variable value (and possibly parameters):

Listing 33. An exemplary script using the ‘Calculator’ function (sending values of a variable instead of the Variable object).

VBScript

```
....
 Function OnRead(val, stat, time)
call Calculator(val) ‘library call
End Function
```

JScript

```
...
function OnRead(val, stat, time)
{
Calculator(val);//library call
}
```

5.16 Remarks Regarding Time

Some methods and fields returning time of a given measured data, alarm, etc. Time is expressed in VT_DATE format. Many ActiveX objects and VBScript language use this format. In the JScript language, it can be used by transferring directly to other object model functions that require supplying time. In order calculations on time in this format in the JScript language to be possible, it should be converted into time format used by a *Date* object of JScript.

If, for example, in JScript language we want to carry out arithmetic calculations on the time of "Var" variable, it can be converted in the following way:

```
var when = new Date(Var.Time);
```

Now we can use all the methods of the *Date* object to carry out calculations (*Date* object's methods are described in JScript documentation). If we want to determine a moment, which precedes the time defined by "when" variable from the above-mentioned example by 20 seconds, we can do it in the following way:

```
when.setTime( when.getTime() -20000 );
```

The functions operating on time built into a specific language can be used freely. Attention should be paid here to transferring time **to** script module. It should be transferred in VT_DATE format. It regards especially the JScript language, in which the use of a variable (within JScript meaning) that includes time or function that returns time (e.g. time as *Date* object), may transfer text representing time instead of time value.

For example, the *Write* method of a *Variable* object called in the following way:

```
var when = new Date();
obiekt.Write( 5,0,when );
```

will cause that text will be transferred instead of time value. In order to transfer time in correct format, use the "getVarDate()" method:

```
obiekt.Write( 5,0,when.getVarDate() );
```

It regards all the methods and fields that require time transfer to script module in the JScript language. In the below example, the use of the "getVarDate" method in the JScript language is useless because the *Message* method of a *Panel* object requires rather a string of characters than time as parameter:

```
var when = new Date();
Asix.Panel.Message( "Now is " + when.getVarDate() );
```

The message output by the *Message* method will have correct form but the same effect can be obtained by executing the following instruction

```
Asix.Panel.Message("Now is " + when );
```

In both the cases, the JScript language converts time into character string and the conversion result and concatenation of strings are transferred to the *Message* method.

In VBScript, additional time conversion before it is transferred to the module is not necessary.

The time used in the SCRIPTS module is the local time. It may lead to an abnormality in case of changing the time into the winter time by one hour. Especially it concerns historical data reading. The *UTCMode* field of a *Variable* object allows to change the way of time presentation in all the fields and parameters (concerning the time) of a *Variable* object method. After this field is set to *true*, the time will be passed as UTC (Coordinated Universal Time). It enables to avoid the discrepancy during changing the time into the winter time - but only when the time of current and archival measurement data is passed as UTC. Notice that JScript „getUTC...." methods will not operate properly if the field has the value *true*. If this field has the value *true*, then the *Status* field of the *Variable* object will have the AS_STAT_UTC_TIME bit set. If the *Status* parameter of the *Write* method of the *Variable* object has the AS_STAT_UTC_TIME bit set, then the *Time* parameter should contain the UTC time.

6 Object Model of Script Module

Below, all the objects made accessible by script module are presented

6.1 Application Object

- Properties
  - Directory Property
  - Name Property
  - Language Property
  - IniFileName Property
- Methods
- Events
- Reserved fields

Access to the *Application* object is enabled via the *Application* field of an *Asix* object.

Properties

Directory Property

Returns the name of start-up directory of the application.

Name Property

Returns the application name. It is the main part of the initialization file name.

Language Property

Returns the language name (e.g. "en" for English).

IniFileName Property

Returns full name of the application's initialization file.

Methods

The object has no methods.

Events

The object generates no events.

Reserved fields

Reserved fields have the following names: Extra, Explicite, Diagnostics, Close, Catalogue, Catalog.

6.2 Alarms Object

- Properties
- Methods
- Cancel Method
- Raise Method
- State Method
- Events
- Reserved Fields

Access to *Alarms* is enabled via the *Alarms* field of an *Asix* object.

Properties

The object has no properties.

Methods

Cancel Method

It should be called in a following way:

object.Cancel(*ident* [, *val1*[, *val2*[, *time* [, *global*]]]])

where:

object	- *Alarms* object;
ident	- alarm identifier;
val1	- alarm parameter; if parameter is omitted, 0 is assumed;
val2	- alarm parameter; if parameter is omitted, 0 is assumed;
Time	- alarm end time; if parameter is omitted, the current time is assumed;
Global	- logical value to determine whether alarm has a global (**true**) or local (**false**) range; if parameter is omitted, **false** is assumed;

The method cancels the alarm with given identifier.

If the method was executed successfully, **true** value is returned. Otherwise **false** is returned.

The *val1* and *val2* are numerical parameters, which are transferred to the Asix alarm system. The values can be pasted into alarm text displayed. It requires appropriate alarm definition (detailed information can be found in the Asix alarm system description).

The *global* parameter has importance for network redundant installations of alarm systems only. Global alarms should be used when the script detecting events is installed on one PC only and its answers are to be accepted regardless of the current system status. Local alarms are used when script is activated on every system station and its answers will only be accepted on the station, which is an active station at the given moment.

Raise Method

It should be called in a following way:

object.Raise(*ident* [, *val1*[, *val2*[, *time*[, *global*]]]])

where:

object	- *Alarms* object;
ident	- alarm identifier;
val1	- alarm parameter; if parameter is omitted, 0 is assumed;
val2	- alarm parameter; if parameter is omitted, 0 is assumed;
Time	- alarm beginning time; if parameter is omitted, the current time is assumed;
Global	- logical value to determine whether alarm has a global (**true**) or local (**false**) range; if parameter is omitted, **false** is assumed.

The method raises the alarm.

If the function was executed successfully, **true** value is returned. Otherwise **false** is returned.

State Method

It should be called in a following way:

object.State(*ident*)

where:

object	- *Alarms* object;
ident	- alarm identifier.

The method returns **true** if alarm is set. Otherwise **false** is returned.

Events

The object generates no events.

Reserved Fields

Reserved fields have the following names: Extra, Explicite, Diagnostics, OnNew.

6.3 Asix Object

- Properties
  - Alarms Property
  - Application Property
  - GlobalData Property
  - Mode Property
  - Panel Property
  - Script Property
  - Scripter Property
  - ThreadGlobalData Property
  - Version Property
- Methods
  - ClearInterval Method
  - CreateEmptyObject Method
  - ExecuteAction Method
  - ExecuteAt Method
  - Parameters Method
  - SetInterval Method
  - Sleep Method
  - Variables Method
- Events
  - OnTerminate Event
- Reserved fields

The*Asix* object is accessible directly by every script executed under control of the Asix script module.

Properties

Alarms Property

Returns an Alarms-type object, which enables access to the Asix alarm system.

Application Property

Returns an Application-type object, which enables access information on the application being executed.

GlobalData Property

Returns a GlobalData-type object, which enables exchange data between scripts. Scripts can be found in different threads.

Mode Property

Returns a number that determines the current operation mode of Asix. The property can have the following symbolic values.

Table 3. Symbolic values of the 'Mode' field of an 'Asix' object.

Symbolic value	Numerical value	Description
Application	2	Asix operates in normal mode (application execution)
Initialize	3	Asix is being initialized
Terminate	4	Asix is completing its operation

Panel Property

Returns a Panel-type object, which makes possible to output information to Asix control panel.

Script Property

Returns a Script-type object, which makes possible to access information on the script being executed at present.

Scripter Property

Returns a Scripter-type object, which makes possible to access information on AsScripter script module.

ThreadGlobalData Property

Returns a ThreadGlobalData-type object, which makes possible to exchange information between scripts in the same thread.

Version Property

The *Version* field returns the Asix version number in the form of *major_version**256 + *minor_version*.

Methods

ClearInterval Method

It should be called in a following way:

object.ClearInterval(*ident*)

where:
object - *Asix* object;
ident - number returned by *SetInterval* or *ExecuteAt* method.

The method cancels the result of execution of the *SetInterval* or *ExecuteAt* method.

CreateEmptyObject Method

The method is designed for scripts in VBScript. It returns an empty object to which the script may add new components. The object allows sending a group of logically connected components (e.g. variables) to other script by means of a *GlobalData* object. The object is a collection of its components.

ExecuteAction Method

It should be called in a following way:

object.ExecuteAction(*action*)

where:
object - *Asix* object;
action - string of characters transferred to Asix system that determine the action to be executed.

Set of available actions that can be executed by means of the *ExecuteAction* method can be found in the Asix system documentation.

ExecuteAt Method

It should be called in a following way:

object.ExecuteAt(*function, time*)

where:
object - *Asix* object;
function - function to be executed;
time - time at which the function is to be executed.

The method returns identifier, which can be used to cancel the result of execution of this function (*ClearInterval*). After calling the function identified with the *function* parameter, the identifier is not longer valid. If *time* refers to the past, the *function* function will be executed at once (or, to be more precise, after the fragment of the script being execute at present has been completed).

Parameters Method

It should be called in the following way:

object.Parameters(*section_name*)

where:
object - *Asix* object;
section_name - string of characters to define station name.

Parameters method enables to process the application's initialization file. The initialization file section name is given as the parameter. As a result of the call, a collection of *Parameter*-type objects will be returned.

SetInterval Method

It should be called in the following way:

object.SetInterval(*function, interval*[, *phase*])

where:
object - *Asix* object;
function - cyclically called function;
interval - time interval in milliseconds between successive calls of the *function* function; if *interval* is a negative number, the *function* will be executed only once;
Phase - time of offset in milliseconds; the parameter determines the offset time in relation to full multiple of interval (e.g. if *interval* is 1 hour, and *phase* is 15 minutes, the function will be started at 0:15, 1:15, 2:15, etc.); if parameter is omitted, the function will be called with no time compensation.

The method returns identifier, which can be used in the *ClearInterval* function in order to cancel the result of calling the *SetInterval* function. If the *interval* parameter is a negative number, the identifier returned by the *SetInterval* function will be cancelled after the first call of the *function* function.

Sleep Method

It should be called in the following way:

object.Sleep(*miliseconds*)

where:

object	- *Asix* object;
miliseconds	- suspend time of the script execution in milliseconds.

The method makes the script execution be suspended for the time defined in the parameter. During the function, time limitations **are** calculated. During execution of the function, the script **does not** response to events. During the function other scripts operating in the same thread are not executed as well. The *Sleep* method does not affect the functioning of scripts in other threads. The function parameter is time in milliseconds.

Variables Method

It should be called in the following way:

object.Variables(*variable_name[, current]*)

where:

object	- *Asix* object;
variable_name	- string of characters to define variable name; in the end of the variable name, after comma, a single-letter archive code can be entered if the variable is to be used to review an archive;
Current	- *l*ogical value to determine whether access to the current values of the variable is required. The parameter defines the operation of the *Variables* method when there is no variable defined with the *variable_name* parameter in the ASMEN's variable database. If **true** was given, the variable must be is the ASMEN's variable database. If it is missing, the *Variables* method will end with failure – the *Variable* object will not be created. If **false** is given, the variable may be absent in database of ASMEN, however it must be present in the ASPAD module archive. In such a case, *variable_name* must include archive type code.

If parameter is omitted, **true** value is assumed, i.e. access to the current values is required.

The method returns a *Variable* object, which enables to access the current and/or archived measuring data. Each time the method is called with the same arguments is called, a new object referring to the same Asix variable is returned. In the event when the value of *bieżąca* parameter is **true** and there is no variable in database of ASMEN, the module returns an empty object. The method always returns an empty object if the variable is either absent in database of ASMEN or is not archived (it is absent in database of ASPAD). In case of successful creation of a

Variable object, the WithArchive field in an Info object enables to determine whether access to the archival values of the variable is possible.

If the variable is present in the ASMEN's variable database, the *Current* field in the newly created *Variable* object will have the value of **true** and the *Value*, *Status* and *Time* fields will be set, respectively, to the last readout of variable values by ASMEN. The WithArchive field in the Info object will have the value of **true** if the *variable_name* parameter includes archive type code and the variable is archived in the archive of that type.

If it is necessary to provide access to archived values of variables, which do not have their current values, i.e. are absent in the ASMEN's database, the value of the *current* parameter should be **false**. The *Current* field in a *Variable* object created in such a way will permanently contain the **false** value and the *Status* field will initially contain the OPC_QUALITY_BAD value. Only after the *ArcRead* or *ArcReadAt* methods are executed, the *Value*, *Status* and *Time* will be set properly. Such behavior of the *Variable* object is related only to a situation when the variable has no current values. If the variable is present in the ASMEN's variable database, the value of the *current* parameter does not matter. At present, only the ASPAD pattern variables may not be included in ASMEN's variable database.

If creation of a *Variable* object is impossible, the method returns an empty object. Below it is presented how to verify the result of method execution.

Listing 34. An exemplary script presenting the way of werifying the 'Variables' method.

VBScript

```
Set Var = Asix.Variables("Emi_CO2")
If TypeName(Var) = "Nothing" Then
Asix.Panel.Message("Wrong variable name" )
End If
```

JScript

```
var Var= Asix.Variables( "Emi_CO2" );
if ( Var == null )
Asix.Panel.Message("Wrong variable name" );
```

Events

OnTerminate Event

The event is generated upon completion of the application. This event will occur before *OnTerminate* in the *Script* object.

Reserved fields

Reserved fields have the following names: *Test, Freeze, Await, BreakAwait, IsAwaiting, Quit, Extra, Explicite, OnASIXPhase, Diagnostics, Modules, Drivers, ConnectObject, CreateObject, Masks, Pictures, Dump* and *Utils*.

6.4 GlobalData Object

- Properties
- Methods
- Events
- Reserved fields

Access to the *GlobalData* object takes place via the *GlobalData* field of an *Asix* object.

The object is used to transfer data between scripts. In order to exchange data between scripts, select the name of component and then assign a value to it. The components of a *GlobalData* are available for every script, including those that can be found in various threads. The object is a collection of its components.

The object components can be any data, also other objects (including script functions).

Properties

The object has the properties, which were assigned to it by scripts only.

Methods

The object has the methods, which were assigned to it by scripts only.

Events

The object has the events, which were assigned to it by scripts only.

Reserved fields

Reserved fields have the following names: *Extra, Explicite* and *Diagnostics*.

6.5 Info Object

Access to the *Info* object takes place via the *Info* field of a *Variable* object.

Fields

ArcAttrProperty

Contains variable archiving attributes. At present, the field can include a logical sum of the following values:

VARATTR_NOPACK - repeated values of the variable are not packed;
VARATTR_SIM - variable values are simulated;
VARATTR_RESTORE - the last archived value is the initial variable value after the system is started;
VARATTR_ALL - all the values are stored.

ArcCycle Property

Contains variable archiving cycle expressed in seconds.

Arcdt Property

Contains time accuracy of variable recording expressed in seconds.

Arcdx Property

Contains minimum change in variable value, which makes it is saved in archive.

ArchiveType Property

Contains a string, which determines archive type (e.g. "M"), in which the variable is archived.

ArcHorizon Property

Contains a horizon of variable archiving.

Count Property

Contains the number of elements, the variable consists of. For normal variables, the value of field is 1. For array variables, the field determines the number of elements the table.

RefreshPeriod Property

Contains variable refreshing cycle expressed in seconds.

WithArchive Property

Returns the logical value of **true** if access to archive is possible. Otherwise **false** is returned.

Methods

The object has no methods.

Events

The object generates no events.

6.6 Panel Object

- Properties
  - Ident Property
- Methods
  - Error Method
  - Message Method
  - Popup Method
  - Warning Method
- Events
- Reserved Components

Access to the *Panel* object takes place via the *Panel* field of an *Asix* object.

The *Panel* object enables to send information to the **asix**'s control panel.

Properties

Ident Property

Ident is a string of characters to identify the sources of messages output by a *Panel* object. If the field has not a different name, the script module assigns the same value to it as the script name defined in script declaration.

Methods

Error Method

It should be called in a following way:

object.Error(*text[, diag]*)

where:

object	- *Panel* object;
text	- string of characters to define the message displayed;
diag	- if the logical value of the parameter is **true**, the message will be written in the log file of **asix**'s system application only. If the parameter is omitted, **false** value is assumed, i.e. the message will be output to control panel.

The method makes the message be output to the **asix**'s control panel. The message status is "ERROR".

Message Method

It should be called in a following way:

object.Message(*text[, diag]*)

where:

object	- *Panel* object;
text	- string of characters to define the message displayed;
diag	- if the logical value of the parameter is **true**, the message will be written in the log file of **asix**'s system application only. If the parameter is omitted, **false** value is assumed, i.e. the message will be output to control panel.

The method makes the message be output to the **asix**'s control panel. The message status is "NORMAL".

Popup Method

It should be called in a following way:

object.Popup(*text*)

where:

object	- *Panel* object;
text	- string of characters to define the message displayed.

The method opens the info window displaying a text defined in the *text*. parameter.

Warning Method

It should be called in a following way:

object.Warning(*text[, diag]*)

where:

object	- *Panel* object;
text	- string of characters to define the message displayed;
diag	- if the logical value of the parameter is **true**, the message will be written in the log file of **asix**'s system application only. If the parameter is omitted, **false** value is assumed, i.e. the message will be output to control panel.

The method makes the message be output to the **asix**'s control panel. The message status is "WARNING".

Events

The object generates no events.

Reserved Components

Reserved components have the following names: *Extra, Explicite and Diagnostics.*

6.7 Parameter Object

- Fields
  - Default Property
  - Name Property
  - Value Property
- Methods
- Events
- Reserved Fields

The *Parameter* object represents a parameter located in the initialization file section or the parameter transferred to the script in script declaration. In last case, the *Name* field can be empty. The *Parameter* object is a component of collection, which is a *Parameters* object.

Fields

DefaultProperty

Value is a default property.

NameProperty

It is a string of characters defining the parameter name.

ValueProperty

It is a string of characters defining the parameter value.

Methods

The object has no methods.

Events

The object generates no events.

Reserved Fields

The object does not allow adding new components.

6.8 Parameters Object

- Properties
- Methods
- Events
- Reserved Fields

The object is a collection of *Parameter* objects representing parameter section of the initialization file or parameters transferred to the script in script declaration. Parameters are put to the initialization file or script declaration in the order they appear. The *Parameters* object is returned by the *Parameters* method in an *Asix* object and by the *Arguments* field in a *Script* object.

Properties

The object has properties of collection, i.e. the *Count* field, which determines number of parameters within collection.

Methods

The object has methods of collection, i.e. the *Item* method that provides access to individual elements of the collection by the use of element number. The elements are numbered beginning from 1.

Events

The object generates no events.

Reserved Fields

Reserved fields have the following names: *Extra, Explicite and Diagnostics.*

6.9 Script Object

- Properties
  - Arguments Property
  - File Property
  - Ident Property
  - Name Property
  - Parameters Property
  - Timeout Property
- Methods
- Events
  - OnTerminate Event
- Reserved Fields

Access to the *Script* object takes place via the *Script* field in an *Asix* object.

Properties

ArgumentsProperty

Returns a *Parameters* object-collection, which contains parameters passed to script declaration.

FileProperty

Returns the name of the file that contains script program.

IdentProperty

Returns the script's numerical identifier.

NameProperty

Returns the script name.

ParametersProperty

It is a synonym of the *Arguments* property.

Timeout Property

Returns the current value of time limit for script execution in milliseconds. Changing this field sets a new limit value. At the same time counting script execution time restarts. Its value is the time expressed in milliseconds. Changing the property can be used in the event when it is necessary to execute activities that require longer time. When 0 is entered, the script execution time control is disabled.

Methods

The object has no methods.

Events

OnTerminate Event

The event is generated upon completion of the script operation. The event will occur after the *OnTerminate* event in the *Asix* object if the script will be completing its operation together with the **asix**'s system application.

Reserved Fields

Reserved fields have the following names: *Extra, Explicite, Dump* and*Diagnostics*.

6.10 Scripter Object

- Fields
  - Version Property
- Methods
- Events
- Reserved Fields

Access to the *Scripter* object takes place via the *Scripter* field in an *Asix* object.

Fields

Version Property

Returns the number of script module version in the following form:

*major_version**16777216+*minor_version**65536+*release*
(or (*major_version* <<24)+(*minor_version*<<16)+ *release*).

Methods

The object has no methods.

Events

The object generates no events.

Reserved Fields

Reserved fields have the following names: *Extra, Explicite, Diagnostics* and*Manager*.

6.11 ThreadGlobalData Object

- Properties
- Methods
- Events
- Reserved Fields

An access to the *ThreadGlobalData* object takes place via the *ThreadGlobalData* field in an *Asix* object.

Similarly to *GlobalData*, this object is used to exchange data between scripts. In case of the *ThreadGlobalData* object, data exchange regards only the scripts from the same thread. The object is a collection of its components.

Properties

The object only has the properties, which were assigned to it by scripts.

Methods

The object only has the methods, which were assigned to it by scripts.

Events

The object only has the events, which were assigned to it by scripts.

Reserved Fields

Reserved fields have the following names: *Extra, Explicite and Diagnostics.*

6.12 Variable Object

- Properties
  - Default Property
  - Current Property
  - Info Property
  - Name Property
  - Overruns Property
  - Status Property
  - Time Property
  - UTCMode Property
  - Value Property
- Methods
  - ArcParam Method
  - ArcRead Method
  - ArcReadAt Method
  - ArcSeek Method
  - ArcTell Method
  - Read Method
  - Write Method
- Events
  - OnRead Event
- Reserved Fields

The object enables to access the process variable values. The object is returned by the *Variables* method in an *Asix* object.

Properties

Default Property

Default property of a *Variable* object is the value of variable, i.e. the value returned by the *Value* field.

Current Property

Returns **true** logical value if the *Value*, *Status*, and *Time* fields are related to the current measuring data and **false** if they are related to archival values. If **true** is entered into this field, the *Value*, *Status* and *Time* properties are updated with the current measured values acquired from ASMEN and the *OnRead* event generation is resumed. If the *ArcRead* and *ArcReadAt* functions are executed, the field is set to **false**. The *OnRead* event is not generated if the field is set to **false**. Entering **false** into the *Current* field has no meaning.

REMARK If the variable has no current values but only the archival ones, i.e. it is not present in the ASMEN module's database, entering **true** into the *Current* field is treated as an error.

Info Property

The field returns an Info object containing the information on the variable.

Name Property

Returns the string that determines the variable name as it was declared in the ASMEN module's database.

Overruns Property

Returns the number, which defines how many times the *Value*, *Status* and *Time* fields have not been updated due to script was busy. Reading this property resets it. Non-zero value of the property shows too slow operation of the script in relation to refreshing frequency and the number of variables handled.

Status Property

Returns a number that determines the variable value status. OPC statuses are used (see: Data Status).

Time Property

Returns a timestamp related to a variable value. The time is the local time.

UTCMode Property

The field defies the way of time presentation in all the time fields and parameters of a Variable object. If this field is *false*, the time is the local time. If the UTCMode field has the value *true*, the time is as UTC (Coordinated Universal Time). *False* is the default value of this field. Notice that JScript „getUTC...." methods will not operate properly if the field has the value *true*. The UTCMode field may be used from the SCRIPTS module v 1.02.000.

Value Property

Returns the variable value. It can be either the current value of variable or archived value depending on the value returned by the *Current* field.

Methods

ArcParam Method

It should be called in the following way:

object.ArcParam([*param*])

where:

object	- *Variable* object;
param	- defines the method of interpolation for ArcReadAt function:
INTR_LINEAR	-linear interpolation;
INTR_PREV	-interpolation with previous value;
INTR_NEXT	-interpolation with next value;
INTR_NEAREST	-interpolation with nearest value.

The method returns 0 if it has been completed successfully or a negative value in case of an error.

ArcRead Method

It should be called in the following way:

object.ArcRead([*type*])

where:

object	- *Variable* object;
type	- direction of reading in relation to current position in archive data:
READ_CURR	-the current data will be read;
READ_NEXT	-the next data will be read.

If *type* is omitted, the READ_NEXT value is assumed.

The method returns 0 if it has been completed successfully or a negative value in case of an error. The method may also return positive values, which meanings are as follows:

STAT_NoData	- no data in archive;
STAT_EarlierTime	- reading trial before the archive beginning;
STAT_LaterTime	- reading trial after the end of archive;
STAT_Break	- gap in archive;
STAT_NotReady	- the data have not been retrieved yet.

The method sets the *Value*, *Status* and *Time* fields to the values read from the archive.

ArcReadAt Method

It should be called in the following way:

object.ArcReadAt(*time*)

where:

object	- *Variable* object;

time - defines date and time (total time).

The method returns the values such as the ArcRead function. The method sets the *Value*, *Status* and *Time* fields to the values read from the archive. If there is no data from the moment defined by the *time* parameter in the archive, the *Value* field will be set on the basis of interpolation of their values saved in the archive. The way of this interpolation is defined with the ArcParam method. If ArcParam has not been called before, the result of the *ArcReadAt* method call is not defined. After the *ArcReadAt* method has been completed successfully, the *Time* field always contains time equal to the time defined in the *time* parameter.

ArcSeek Method

It should be called in the following way:

object.ArcSeek(*time*[, *start_point*])

where:

object - *Variable* object;

time - defines a new position of archive index and is the total time in case of ORIGIN_SET.; total time is expressed in VT_DATE format; in other cases, time is a numerical value, which defies offset in seconds referred to the current position of archive index (for ORIGIN_CUR) or to the end of archive (for ORIGIN_END);

start_point - the parameter defines the time point to which the *time* parameter is related to and can accept the following values:
ORIGIN_SET
ORIGIN_CUR
ORIGIN_END
If the parameter is omitted it is assumed its value is ORIGIN_SET and the *time* parameter defines total time.

The method returns 0 if it has been completed successfully or a negative value in case of error. Additional information on time can be found in "Remarks Concerning Time..

This method does not update the *Value*, *Time* and *Status* fields. In order to update the fields, the *ArcRead* method should be performed.

ArcTell Method

It should be called in the following way:

object.ArcTell

where:

Object - *Variable* object.

The method returns time the archive index is set to.

Read Method

It should be called in the following way:

object.Read( )

where:
object - *Variable* object.

The method reads values from external device or the variable put in the NONE channel. After this method has been completed, the *Value*, *Status* and *Time* fields will be updated respectively. If the fields were used to review variable archive before, after *Read* function has been completed, they will relate to the current values, i.e. the *Current* field will be set to **true**.

The method returns 0 if the write operation has been completed successfully.

Write Method

It should be called in the following way:

object.Write(*value*[, *status*[, *time*]])

where:
object - *Variable* object;
value - value to be entered;
status - status of the value being entered; if the parameter is omitted, the "correct data" status is assumed. OPC status is used (see: Data Status);
time - timestamp of the value being entered; if the parameter is omitted, the current time is assumed.

The method enables to write values to external device or variable put in the NONE channel. The *status* and *time* parameters have the meaning only for variables in the NONE channel, for which ***Writing data and status*** parameter must be additionally determined in the NONE driver channel of the XML application file.

The method returns 0 if the write operation has been completed successfully.

The *Write* method does not update the *Value*, *Time* and *Status* fields. The fields can be updated as a result of execution of *Read* method or as a result of automatic readout of new value by ASMEN (if the fields relate rather to the current values than the archival ones, i.e. if the *Current* field is set to **true**).

Events

OnRead Event

The event is generated when ASMEN has read a new measured value. The form of event response function should be as follows:

Listing 35. An exemplary script with the function of reaction on a 'Variable' object event.

VBScript	JScript
Function *function_name*(*value*, *status*, *time*) .... **End Function**	**function** *function_name*(*value*, *status*, *time*) { .... }

parameters of the function define the value, status and time of measured data, respectively. OPC status is used (see: Data Status)

The event is generated only if the *Current* field is set to **true**.

Reserved Fields

Reserved fields have the following names: Extra, Explicite, Diagnostics, First, Last, Next, Previous, WithBreaks, SkipBreaks, NextBreak, PrevBreak, FirstBreak, LastBreak, Break, BreakEnd, Find, Search, Info, AtBreak, Move, AtData, Archive, MoveTo, Refresh, Open, ArcParam.

7 Configuration of AsScripter Module

In **asix5** Script module is configured with use of Architect module.

See: Architect user's manual, chapter *3.14. Configuration of Script and Action Parameters*.

8 SCRIPT Operator Action

The script can also be started by the use of the SCRIPT operator action.

The action syntax is as follows.

R*SCRIPT filename, script_parameters execution_parameters*

Abbreviation - **SCR**
Operation type - command to execute a script from a text file.
Parameter *file_name*:
Meaning - name of file containing the script. If any access path is not given then the directories of diagram paths (MASK_PATH) are searched. If any file extension is not given, then the **vbs** is added by default.
Type - text.
Parameter *script_parameters*:
Meaning - parameters transferred to the executed script; is identical as in case of scripts activated via the ***Scripts*** parameter;
Parameter *execution_parameters*:
Meaning - is identical as in case of scripts activated via the ***Scripts*** parameter (see: 7. *Configuration of AsScripter Module)*.

9 New Functions for Asix5

9.1 Introduction

AsScripter in version 1.7 has been enhanced with several functions allowing the loading of series of archival data, and also to read variable attributes from the variable database and to access information on the current **asix** system user.

9.2 Loading data series from Aspad archive

Access to the archival data series is performed with the use of the **asix** Connect module. The object Variable has been enhanced with 2 functions analogous to the functioning of the archival data access Automation available in the Asix Connect.

9.2.1 ReadRaw – Reading raw data

Reading a series of raw archival data is executed with use of the ReadRaw method of Variable object:

object.ReadRaw (*start_time*, *end_time*, *data*)

ReadRaw reads values, statuses and timestamps of the variable from the archive for the given period. It reads real values stored in the archive and also returns the values constituting the ends of the range.

Start_time and *end_time* parameters must be of a Date or String type and are local or UTC times, depending on the setting of UTCMode field of the object. In the case of String type, it is assumed that the time is given in compliance with the rules determined for OPC relative time (9.2.3).

When the reading operation is over, the *data* parameter contains an array of read samples. The array contains as many lines as the number of read samples. Each line includes three elements:

0. value – of Double type; contains source data converted to Double type,
1. timestamp – of Date type, in the convention compliant with UTCMode field setting.

2. quality – of Integer type; contains 32-bit OPC status of historical data (9.2.4).

The array also returns the limit values for the given time period. If the archive does not store a sample registered accurately in *start_time*, the last sample before this time is returned. If the archive does not store a sample registered accurately in *end_time*, the first sample after this time is returned. If the limit values cannot be read from the archive, a sample with the quality field of OPCHDA_NOBOUND value is returned.

Example of use

In the following example, raw data of D archive for Var variable for the period from StartTime to EndTime are read.

```
On Error Resume Next
Err.Number = 0

Set VarArc = Asix.Variables( "Var,D" )
VarArc.ReadRaw StartTime, EndTime, RawData
Panel.Message "ReadRaw " & StartTime & ", " & EndTime & ", RawData"
If Err.Number = 0 Then
   Panel.Message "RawData(" & LBound(RawData, 1) & "-" & UBound(RawData, 1) & ", " _
      & LBound(RawData, 2) & "-" & UBound(RawData, 2) & ")"
   For i = LBound(RawData, 1) To UBound(RawData, 1)
      Panel.Message "   Value=" & RawData(i, 0) & ", Time=" & RawData(i, 1) & ", Quality=" _
         & Hex(RawData(i, 2))
   Next
Else
   Panel.Message "Error # " & Hex(Err.Number) & " " & Err.Description
End If
```

9.2.2 ReadProcessed – Reading aggregated data

Reading a series of aggregated archival data is executed with use of the ReadProcessed method of Variable object:

object.ReadProcessed (*start_time*, *end_time*, *aggregate_name, aggregation_interval, data*)

ReadProcessed calculated aggregates of the given variable for the given period.

The meaning of *start_time* and *end_time* parameters is the same as in ReadRaw method.

Aggregation_interval parameter should contain the duration of the aggregation interval given as Date or String type (OPC relative time). The aggregation interval will always be calculated from the beginning of the day in UTC time and should not contain D, MO, Y components.

Aggregate_name parameter should contain one of the following aggregate names.

English name	Polish name	Aggregate calculation method
Start	*Początek*	Value for the start of the interval. Timestamp is the interval start timestamp.
End	*Koniec*	Value for the end of the interval. Timestamp is the interval end timestamp.
Delta	*Przyrost*	Difference of values at the end and at the beginning of the interval.
Min	*Min*	Minimal value in the interval
Max	*Max*	Maximum value in the interval
Range	*Zakres*	Difference between the maximum and minimum value in the interval.
Total	*Suma*	Sum of time-weighed values in the interval (integral after the time).
Average	*Średnia*	Average of time-weighed values in the interval.
Average0	*Średnia0*	Average of time-weighed values in the interval. In the period, where the value for calculation is unavailable, value of 0 is used.
Quality_Good	*Jakość_Dobra*	Percent of samples of good quality in the interval (1 = 100%).
Quality_Uncertain	*Jakość_Niepewna*	Percent of samples of uncertain quality in the interval (1 = 100%).
Quality_Bad	*Jakość_Zła*	Percent of samples of bad quality in the interval (1 = 100%).

Example of use

In the following example, 5-minute averages of D archive for Var variable for the period from StartTime to EndTime are read.

```
On Error Resume Next
Err.Number = 0

Set VarArc = Asix.Variables( "Var,D" )
VarArc.ReadProcessed StartTime, EndTime, "Average", "5M", ProcData
Panel.Message "ReadProcessed " & StartTime & ", " & EndTime & ", Average, 5M, ProcData"
If Err.Number = 0 Then
   Panel.Message "ProcData(" & LBound(ProcData, 1) & "-" & UBound(ProcData, 1) & ", " _
      & LBound(ProcData, 2) & "-" & UBound(ProcData, 2) & ")"
   For i = LBound(ProcData, 1) To UBound(ProcData, 1)
      Panel.Message "   Value=" & ProcData(i, 0) & ", Time=" & ProcData(i, 1) _
         & ", Quality=" & Hex(ProcData(i, 2))
   Next
Else
   Panel.Message "Error # " & Hex(Err.Number) & " " & Err.Description
End If
```

9.2.3 OPC time format

OPC relative time format has the following syntax:
keyword +/- offset +/- offset ...

Possible keyword and offset values are presented in the table below. Spaces and tabulators are ignored. Each offset parameter must be preceded with an integer number specifying its multiplicity and direction.

Keyword	Description
NOW	Current time
SECOND	Current second start
MINUTE	Current minute start
HOUR	Current hour start
DAY	Current day start
WEEK	Current week start
MONTH	Current month start
YEAR	Current year start

Offset	Description
S	Time offset in seconds
M	Time offset in minutes
H	Time offset in hours
D	Time offset in days
W	Time offset in weeks
MO	Time offset in months
Y	Time offset in years

For instance, the record DAY -1D+7H30M could represent the start time of data for a daily report generated on the current day (DAY = the first timestamp of the current day). The record -1D signifies the first timestamp of yesterday, +7H is 7:00 hour yesterday, +30M is the hour 7:30 yesterday; the character + in the last offset is moved from the previous offset).

Similarly, MONTH-1D+5h means 5:00 hour of the last day of the previous month, NOW-1H15M is 1 hour and 15 minutes ago, and YEAR+3MO is the 1st of April of the current year.

This format also allows the length of the time period to be expressed. In such a case, skip the first keyword part in the described format.

9.2.4 OPC statuses for historical data

Historical data statuses are produced by a combination of the basic 16-bit status in the 16 younger bits and the 16-bit field in the 16 older bits. In the following table, the meaning of 16 older bits is given.

Symbolic value	Hexadecimal code
OPCHDA_EXTRADATA	0x00010000
OPCHDA_INTERPOLATED	0x00020000
OPCHDA_RAW	0x00040000
OPCHDA_CALCULATED	0x00080000
OPCHDA_NOBOUND	0x00100000
OPCHDA_NODATA	0x00200000
OPCHDA_DATALOST	0x00400000
OPCHDA_CONVERSION	0x00800000
ASKOM_HDA_ARCHIVE_END	0x01000000

The meaning of the first eight bits complies with OPC specification, and ASKOM_HDA_ARCHIVE_END bit means that the sample is the last one currently available in the archive.

9.3 ReadAttribute – reading variable attributes from variables database

The variable object provides access to the variable attributes stored in the variable database with the use of ReadAttribute method.

object.ReadAttribute(*attribute_name*)

For *object* variable, the method returns the value of the given attribute.

As the *attribute_name* parameter, give the attribute name or one of the constants presented in the table.

Constant name	Meaning
atrDescription	Variable description
atrEU	Unit of measurement
atrRangeLo	Lower measurement range
atrRangeHi	Upper measurement range
atrSampleRate	Sampling rate
atrArchiveRate	Archiving period

Example of use

```
Set VarObj = Asix.Variables("Var,D")
VarName = VarObj.ReadAttribute("Nazwa")
VarChan = VarObj.ReadAttribute("Kanał")
VarFun = VarObj.ReadAttribute("Funkcja przeliczająca")
VarDescr = VarObj.ReadAttribute(atrDescription)
```

9.4 Access to information on the current Asix system user

Information on the current user of Asix system is available as the new fields of Asix object:

object.CurrentUserId – current user identifier
object.CurrentUserName – current user name
object.CurrentUserLevel – current user authorization level

10 Listings