AxxonSoft

INTELLECT™ Software Package

Programming Guide (JavaScript)

Version 1.4

Moscow 2010

axxon
Video Surveillance and Security Solutions

Contents

1 Programming in JavaScript

1.1 JavaScript functionality in the Intellect software package

The JavaScript programming language is used in the Intellect software package to implement additional user functions not included in the basic Intellect functionality.

JavaScript is a de-facto standard for developing and running user scripts. Intellect supports the version of JavaScript based on ActiveX technology by Microsoft. The general description of the JavaScript object model is given in the Microsoft documentation (for example, MSDN).

The JavaScript scripts in Intellect are executed using the standard ActiveX modules included in the Windows operating system. So, any objects from the ActiveX-based JavaScript can be used in developing the scripts for Intellect.

A set of specialized JavaScript objects is provided in Intellect for handling Intellect system objects, and for sending and receiving system events and actions.

1.2 Description of the JavaScript object model in Intellect

1.2.1 The Core object and its built-in methods

1.2.1.1 *The Core object*

The Core object is a global static object providing the methods for monitoring and controlling the Intellect system objects. Core methods allow receiving information about the existing objects, generating actions for them and changing their states. Additional Core objects can pause script execution, script debugging, creating and calling global variables.

Core is not a prototype, thus no other objects can be created based on it (it cannot be used as a template). All Core methods are static. Thus, Core methods are called directly from the script with no need for a Core prefix.

1.2.1.2 *The SetObjectParam method*

The SetObjectParam method sets the values of object parameters.

Method call syntax

```
function  SetObjectParam(objtype: String,  id: String,  param : String, value : String)
```

Method arguments

objtype

Required argument. The type of the object whose parameters are to be set. Takes the following values: Type – String, range – existing object types.

id

Required argument. Identification number of the object of the type set in the objtype parameter. Takes the following values: Type – String, range – existing object identification numbers of the specified type.

param

Required argument. The parameter of the object. Takes the following values: Type – String, range – available parameters of the object.

value

Required argument. The value to be set for the parameter specified in the param argument. Takes the following values: Type – String, range – depends on the parameter.

Usage examples

Problem. When Macro 1 starts, check if Cameras 1 to 4 are set to color image transmission. If a camera is set for black-and-white image transmission, switch it to color mode (parameter - color, value – true (1)).

```
if (Event.SourceType == "MACRO" && Event.SourceId == "1"  && Event.Action == "RUN")
{
 var i;
 for(i=1; i<=4; i=i+1)
 {
  if (GetObjectParam("CAM", i , "color") == "0")
  {
   SetObjectParam("CAM", i, "color", "1");
  }
 }
}
```

1.2.1.3 *The SetObjectState method*

The SetObjectState method changes the state of objects.

Method call syntax

```
function  SetObjectState(objtype : String, id : String, state : String)
```

Method arguments

objtype

Required argument. The type of the object whose state is to be changed. Takes the following values: Type – String, range – existing object types.

id

Required argument. Identification number of the object of the type set in the objtype parameter. Takes the following values: Type – String, range – existing object identification numbers of the specified type.

state

Required argument. The state to switch the object to. Takes the following values: Type – String, range – available states of the object.

Usage examples

Problem. Check if Camera 1 is armed every hour. If Camera 1 is disarmed, arm it.

Note. The Timer object with identification number 1 should be created beforehand. Set the Minutes parameter of the Timer object to 30.The timer would go off every hour at half past the hour - 09:30, 10:30, 11:30, etc.

```
if (Event.SourceType == "TIMER" && Event.SourceId == "1" && Event.Action == "TRIGGER")
{
 if (GetObjectState("CAM", "1") == "DISARMED")
 {
  SetObjectState("CAM", "1", "ARMED");
 }
}
```

1.2.1.4 *The DebugLogString method*

The DebugLogString method outputs the user messages into the debug windows of the Editor-Debugger utility.

Method call syntax

```
function  DebugLogString(output : String)
```

Method arguments

output

Required argument. The text message to be displayed in the debug window of the Editor-Debugger utility. Takes the following values: Type – String.

Usage examples

Problem. Output to the debugger window all microphone events registered by the system.

```
if (Event.SourceType == "OLXA_LINE")
{
 var msgstr = Event.MsgToString();
 DebugLogString("Event from the microphone " + msgstr);
```

```
}
```

1.2.1.5 *The Sleep method*

The Sleep method pauses the execution of the script for a specified period of time.

Method call syntax

```
function  Sleep(milliseconds : int)
```

Method arguments

milliseconds

Required argument. The length of time that the script will be inactive for. Set in milliseconds. Takes the following values: Type – int.

Usage examples

Problem. When Macro 1 starts, play the following audio files one by one: cam_alarm_1.wav, cam_alarm_2.wav, cam_alarm_3.wav from the …\Intellect\Wav\ folder. Set a 5 seconds (5000 milliseconds) delay before starting each subsequent file.

```
if (Event.SourceType == "MACRO" && Event.SourceId == "1" && Event.Action == "RUN")
{
var i;
 for(i=1; i<=3; i=i+1)
 {
   DoReactStr("PLAYER", "1", "PLAY_WAV", " file<\cam_alarm_"  + i + ".wav>");
   Sleep(5000);
 }
}
```

1.2.1.6 *The Itv_var method*

The Itv_var method sets and returns the values of global variables.

Method call syntax

```
function  Itv_var (globalvar : String) : String
```

Method arguments

globalvar

Required argument. The name of the global variable. Takes the following values: Type – String, satisfying the rules for the names of the string parameters in the Windows registry.

Additional information

Global variables are stored in the Windows registry to maintain their values after Windows restart. All global variables are stored in the registry branch HKEY_USERS\S-1-5-21-…\Software\ITVScript\ITVSCRIPT и HKEY_CURRENT_USER\Software\ITVScript\ ITVSCRIPT. To access a global variable directly from the registry, search the registry for it by its name.

Usage examples

Problem. When Macro 1 starts, save the value of the bright parameter of Camera 10 to the cam10bright global variable. When Macro 2 starts, set the bright parameter of Cameras 1 to 4 to the value of the cam10bright global variable.

```
if (Event.SourceType == "MACRO" && Event.Action == "RUN")
{
 if(Event.SourceId == "1")
 {
  Itv_var("cam10bright") = GetObjectParam("CAM", "1", "bright");
 }
 if (Event.SourceId == "2")
 {
  var cam10bright = Itv_var("cam10bright");
  for(i=1; i<=4; i=i+1)
  {
   SetObjectParam("CAM", i, "bright", cam10bright);
  }
 }
}
```

1.2.1.7 *The GetObjectParentType method*

The GetObjectParentType method returns the type of the parent object of the current object according to the object hierarchy.

Method call syntax

```
function  GetObjectParentType (objtype : String) : String
```

Method arguments

objtype

Required argument. The type of the object whose parent's type should be returned. Takes the following values: Type – String, range – existing object types.

Additional information

The Main object is the highest level object in the hierarchy. It is the parent for all objects of the Computer, Screen, and other types.

Usage examples

Problem. When Macro 1 starts, display in the debugger window the names of four object types of higher hierarchical order starting from the detection zone.

```
if (Event.SourceType == "MACRO" && Event.SourceId == "1" && Event.Action == "RUN")
{
 var objtype = "CAM_ZONE";
 DebugLogString(objtype);
 for(var i = 1; i<=4; i=i+1)
 {
  objtype = GetObjectParentType(objtype);
  DebugLogString(objtype);
 }
}
```

1.2.1.8 *The GetIPAddress method*

The GetIPAddress method returns the IP-address of the Intellect kernel according to current video surveillance system architecture.

Method call syntax

```
function  GetIPAddress (dst : String, src : String) : String
```

Method arguments

dst

Required argument. The name of the remote computer where the Intellect kernel is installed. The value of dst should correspond to one of the names of the computers registered during setup of the video surveillance system. Takes the following values: Type – String, meeting the requirements for network computer names; range – computer names existing in the system.

src

Required argument. The name of the local computer where the script executes. The value of src should match the name of the local computer as it is registered in Intellect. Takes the following values: Type – String; meeting the requirements for network computer names.

Additional information

The information about all connections of the local computer (kernel) to other remote computers (kernels) registered during the setup of the distributed architecture, is displayed in the Architecture tab of the System Settings window.

Usage examples

Problem. Upon a camera alarm, determine the name of the server this camera is connected to, and output the IP-address of the connection between this server and the local computer where the script executes, to the debugger window.

```
if (Event.SourceType == "CAM" && Event.Action == "MD_START")

{

 var camid = Event.SourceId;

 var compname = GetObjectParentId("CAM", camid, "COMPUTER");

 var ip = GetIPAddress("WS1","WS1"); \\if the script is run on the computer where kernel of  Intellect software has been installed

 DebugLogString("IP-address of the alarmed camera computer" + ip);

}
```

1.2.1.9 *The GetObjectName method*

The GetObjectName method returns the name of the object that it was given upon creation.

Method call syntax

```
function  GetObjectName(objtype : String, id : String) : String
```

Method arguments

objtype

Required argument. The type of the object whose name is to be returned. Takes the following values: Type – String, range – existing object types.

id

Required argument. Identification number of the object of the type set in the objtype parameter. Takes the following values: Type – String, range – existing object identification numbers of the specified type.

Usage examples

Problem. In case of alarm in any sensor, open the information window with the following text - "Alarm in the <alarmed sensor name> sensor connected to the <server name which the sensor is connected to> server".

Note. Create the information dialog window beforehand using the Arpedit.exe utility and save it as test.dlg in the …\Intellect\Dialog folder.

```
if (Event.SourceType == "GRAY" && Event.Action == "ALARM")

{

 var grayid = Event.SourceId;

 var grayname = GetObjectName("GRAY", grayid);

 var compname = GetObjectParentId("GRAY", grayid, "COMPUTER");

 DoReactStr("DIALOG", "test", "CLOSE_ALL","");

 DoReactStr("DIALOG", "test", "RUN","Alarm in the ‘" + grayname + "’ sensor connected to the ‘" + compname + "’ server.");

}
```

1.2.1.10 *The GetObjectState method*

The GetObjectState method returns the state of the object at the moment of method call.

Method call syntax

```
function  GetObjectState(objtype : String, id : String) : String
```

Method arguments

objtype

Required argument. The type of the object whose state is to be returned. Takes the following values: Type – String, range – existing object types.

id

Required argument. Identification number of the object of the type set in the objtype argument. Takes the following values: Type – String, range – existing identification numbers of the objects of the specified type.

Usage examples

Problem. When Relay 1 activates (for example, on pressing the button connected to Relay 1), arm Sensor 1. The next time Relay 1 activates, disarm Sensor 1.

```
if (Event.SourceType == "GRELE" && Event.SourceId == "1" && Event.Action == "ON")

{
```

```
 if(GetObjectState("GRAY", "1")=="DISARM")
 {
  SetObjectState("GRAY", "1", "ARM");
 }
 else
 {
  SetObjectState("GRAY", "1", "DISARM");
 }
}
```

1.2.1.11 *The GetObjectParam method*

The GetObjectParam method returns the value of the specified parameter of the object at the moment of method call.

Method call syntax

```
function  GetObjectParam(objtype : String, id : String, param : String) : String
```

Method arguments

objtype

Required argument. The type of the object whose parameter's value is to be returned. Takes the following values: Type – String, range – existing object types.

id

Required argument. Identification number of the object of the type set in the objtype argument. Takes the following values: Type – String, range – existing identification numbers of the objects of the specified type.

param

Required argument. The name of the parameter whose value is to be returned. Takes the following values: Type – String, range – available parameters of the object.

Usage examples

See the example for the SetObjectParam method.

1.2.1.12 *The GetObjectParentId method*

The GetObjectParentId method returns the identification number of the parent of the specified object.

Method call syntax

```
function  GetObjectParentId(objtype : String, id : String, parent : String) : String
```

Method arguments

objtype

Required argument. The type of the object whose parent's identification number should be returned. Takes the following values: Type – String, range – existing object types.

id

Required argument. Identification number of the object of the type set in the objtype argument. Takes the following values: Type – String, range – existing identification numbers of the objects of the specified type.

parent

Required argument. The type of the object which is the parent of the object type specified by the objtype argument. Takes the following values: Type – String, range – existing object types.

Usage examples

Problem. If a camera turns off or stops transmitting a video signal, send an e-mail message with the following subject: "Warning! Camera turned off" and, in the message body, the number of the camera and of the server it is connected to.

Note. The Short Messages Service is supposed to be installed and working properly.

```
if ((Event.SourceId == "CAM" && Event.Action == " DETACH ") || (Event.Action == " REC_STOP "))
{
 var cam_id = Event.SourceId;
 var parent_comp_id = GetObjectParentId("CAM", cam_id, "COMPUTER");
 DoReactStr("MAIL_MESSAGE", "1", " SETUP ", "subject<Warning! Camera turned off>,body<The " + cam_id + " camera connected to the " + parent_comp_id + " server turned off>");
  DoReactStr("MAIL_MESSAGE", "1", "SEND", "");
}
```

1.2.1.13 *The DoReactSTr method*

The DoReactStr method generates the response actions for the objects. It sends the action to the specified object. The action is transferred directly to the kernel where the object belongs, and not to the whole system. The action is specified as a group of String arguments.

Method call syntax

```
function DoReactStr(objtype : String, id : String, action : String, param<value> [, param<value>] : String)
```

Method arguments

objtype

Required argument. The type of the object that the action should be generated for. Takes the following values: Type – String, range – existing object types.

id

Required argument. Identification number of the object of the type set in the objtype argument. Takes the following values: Type – String, range – existing identification numbers of the objects of the specified type.

action

Required argument. The action to be generated. Takes the following values: Type – String, range – available actions for the objects of the specified type.

param<value>

Required argument. Several arguments of this type can be specified. The parameters of the action.

One parameter has the following syntax:

"param<value>", where

param – name of the parameter;

value – value of the parameter.

Several parameters have the following syntax:

"param1<value1>,param2<value2>…"

Elements of the list are separated by commas with no white space. If no parameters need to be specified, an empty string is used:

DoReactStr("CAM","1","MD_START","");

The param argument can take the following values: Type – String, range – available parameters of the specified action. The value argument can take the following values: Type – String, range – depends on the parameter.

Additional information

Two types of system messages are available in the Intellect system: events and actions.

The events usually contain some information and are used as notifications sent to all Intellect kernels connected to each other during the system setup.

The actions are the control commands sent to specific objects. An action is transmitted only to the kernel where the related object belongs, and not to the whole system.

The DoReactStr and DoReact methods are used to generate actions. The NotifyEventStr and NotifyEvent methods are used to generate events.

Usage examples

Problem. When an alarm is received from a camera, switch Monitor 1 to single window mode and show the video from the alarmed camera in this window.

```
if (Event.SourceType == "CAM" &&  Event.Action == "MD_START")

{

 var camid = Event.SourceId;

 DoReactStr("MONITOR","1","ACTIVATE_CAM","cam<"+ camid +">");

 DoReactStr("MONITOR","1","KEY_PRESSED","key<SCREEN.1>");

}
```

1.2.1.14 *The DoReact method*

The DoReact method generates actions for the objects. It sends the action to the specified object. The action is transferred directly to the kernel where the object belongs, and not to the whole system. The action is specified using the MsgObject object.

Method call syntax

```
function  DoReact(msgevent : MsgObject)
```

Method arguments

msgevent

Required argument. The action sent to the specified object. Takes the following values: MsgObject objects created earlier in the script.

Additional information

Two types of system messages are available in the Intellect system: events and actions.

The events usually contain some information and are used as notifications sent to all Intellect kernels connected to each other during the system setup.

The actions are the control commands sent to specific objects. An action is transmitted only to the kernel where the related object belongs, and not to the whole system.

The DoReactStr and DoReact methods are used to generate actions. The NotifyEventStr and NotifyEvent methods are used to generate events.

Usage examples

Problem. When Relay 1 closes, close Relays 2 and 3. When Relay 1 opens, open Relay 2.

```
if (Event.SourceType == "GRELE" && Event.SourceId == "1")

{

 var msgevent = Event.Clone();
```

```
 if(Event.Action == "ON")
 {
  msgevent.SourceId = "2";
  DoReact(msgevent);
  msgevent.SourceId = "3";
  DoReact(msgevent);
 }
 if(Event.Action == "OFF")
 {
  msgevent.SourceId = "2";
  DoReact(msgevent);
 }
}
```

1.2.1.15 *The DoReactSetupCore method*

The DoReactSetupCore method is used for changing the parameters of the object. It changes only the specified parameters, leaving other parameters intact.

Method call syntax

```
function  DoReactSetupCore(objtype : String,  id : String, param<value>  [, param<value>] : String )
```

Method arguments

objtype

Required argument. The type of the object whose parameters are to be changed. Takes the following values: Type – String, range – existing object types.

id

Required argument. Identification number of the object of the type set in the objtype argument. Takes the following values: Type – String, range – existing identification numbers of the object of the specified type.

param<value>

Required argument. Several arguments of this type can be specified. The parameters of the action.

One parameter has the following syntax:

“param<value>”, where

param – name of the parameter;

value – value of the parameter.

Several parameters have the following syntax:

“param1<value1>,param2<value2>…”.

Elements of the list are separated by commas with no white space.

The param argument can take the following values: Type – String, range – available parameters of the specified action. The value argument can take the following values: Type – String, range – depends on the parameter.

Additional information

Usage examples

Problem. When Macro 1 starts, set the values of the following parameters of Cameras №1-4: PTZ device number (telemetry_id) and synchronous microphone number (audio_id). The values should be equal to the camera numbers plus 1.

```
if (Event.SourceType == "MACRO" && Event.SourceId == "1" && Event.Action == "RUN")
{
 var i;
 for(i=1; i<=4; i=i+1)
 {
  DoReactSetupCore ("CAM", i, "telemetry_id<" + (i+1) +">,audio_id<" + (i+1) + ">");
 }
}
```

1.2.1.16 *The NotifyEventStr method*

The NotifyEventStr method generates system events. Events are sent to all kernels connected to the local kernel. An event is specified as a group of String arguments.

Method call syntax

```
function  NotifyEventStr(objtype : String, id  : String, event : String, param<value>  [, param<value>] : String )
```

Method arguments

objtype

Required argument. The type of the object that the event should be generated for. Takes the following values: Type – String, range – existing object types.

id

Required argument. Identification number of the object of the type set in the objtype argument. Takes the following values: Type – String, range – existing identification numbers of the object of the specified type.

event

Required argument. The event to be generated. Takes the following values: Type – String, range – available events for the objects of the specified type.

param<value>

Required argument. Several arguments of this type can be specified. The parameters of the event.

One parameter has the following syntax:

"param<value>", where

param – name of the parameter;

value – value of the parameter.

Several parameters have the following syntax:

"param1<value1>,param2<value2>…".

Elements of the list are separated by commas with no white space. If no parameters need to be specified, an empty string is used:

DoReactStr("CAM","1","MD_START","");

The param argument can take the following values: Type – String, range – available parameters of the event. The value argument can take the following values: Type – String, range – depends on the parameter.

Additional information

Two types of system messages are available in the Intellect system: events and actions.

The events usually contain some information and are used as notifications sent to all Intellect kernels connected to each other during the system setup.

The actions are the control commands sent to specific objects. An action is transmitted only to the kernel where the related object belongs, and not to the whole system.

The DoReactStr and DoReact methods are used to generate actions. The NotifyEventStr and NotifyEvent methods are used to generate events.

Usage examples

Problem. When an alarm is received, send the “panic lock” event corresponding to the camera region. For camera identification numbers from 1 to 4, use region 1; for camera numbers from 5 to 10, use region 2.

```
if (Event.SourceType == "CAM" && Event.Action == "MD_START")
{
 var regionid;
 if (Event.SourceId <=4)
 {
  regionid = "1";
 }
 if ((Event.SourceId > 4) &&  (Event.SourceId < 10))
 {
  regionid = "2";
 }
 NotifyEventStr("REGION", regionid, "PANIC_LOCK", "");
}
```

1.2.1.17 *The NotifyEvent method*

The NotifyEvent method generates system events. The event is sent to all kernels connected to the local kernel. The event is specified using the MsgObject object.

Method call syntax

```
function  NotifyEvent(msgevent : MsgObject)
```

Method arguments

msgevent

Required argument. The event sent to the system. Takes the following values: MsgObject objects created earlier in the script.

Additional information

Two types of system messages are available in the Intellect system: events and actions.

The events usually contain some information and are used as notifications sent to all Intellect kernels connected to each other during the system setup.

The actions are the control commands sent to specific objects. The actions are transmitted only to the kernel where the object belongs, and not to the whole system.

The DoReactStr and DoReact methods are used to generate actions. The NotifyEventStr and NotifyEvent methods are used to generate events.

Usage examples

Problem. When the Active Archive 1 module starts archiving video recordings, turn off the Analog Out 1 of Video Capture Card 2. Send the command as an event to be registered in the Events Log.

```
if (Event.SourceType == "ARCH" && Event.SourceId == "1" && Event.Action == "ACTIVE ")
{
 var msgevent = CreateMsg();
 msgevent.SourceType = " GRABBER ";
 msgevent.SourceId = "2";
 msgevent.Action = "MUX1_OFF";
 NotifyEvent(msgevent);
}
```

1.2.1.18 *The CreateMsg method*

The CreateMsg method creates objects based on the MsgObject prototype.

Method call syntax

```
function  CreateMsg() : MsgObject
```

Method arguments

No arguments.

Usage examples

Problem. When an alarm is received, send the “panic lock” event corresponding to the camera region. For camera identification numbers from 1 to 4, use region 1, for camera numbers from 5 to 10, use region 2.

```
if (Event.SourceType == "CAM" && Event.Action == "MD_START")
{
 var msgevent = CreateMsg();
 msgevent.SourceType = "REGION";
 msgevent.Action = "PANIC_LOCK";
 if (Event.SourceId <=4)
 {
```

```
  msgevent.SourceId = "1";
 }
 if ((Event.SourceId > 4) &&  (Event.SourceId < 10))
 {
  msgevent.SourceId = "2";
 }
 NotifyEvent(msgevent);
}
```

1.2.2 The MsgObject and Event objects and their built-in methods and properties

1.2.2.1 *The MsgObject and Event objects*

MsgObject is a prototype (template) used to create objects with methods and properties for handling events. The methods and properties of MsgObject allow receiving the information about objects that send or receive events, generating actions, changing object states, etc.

The methods and properties of the MsgObject prototype can be called via its instance objects or the Event static object.

Event is a static object used to call events in Intellect. The Event object represents the system event that launched the script. All MsgObject methods and properties are available for the Event object.

The CreateMsg method of the Core object is used to create instances based on the MsgObject prototype.

1.2.2.2 *The GetSourceType method*

The GetSourceType method returns the type of the MsgObject or Event object.

Method call syntax

```
function  GetSourceType() : String
```

Method arguments

No arguments.

Usage examples

Problem. When Macro 1 starts, activate Detection Zones *.1 in the Day mode for Cameras № 1 – 4. When Macro 2 starts, activate Detection Zones *.2 in the Night mode for Cameras № 1 – 4. When Macro 3 starts, activate Detection Zones *.3 in the Rain mode for Cameras № 1 – 4.

Note. Symbol "" corresponds to identification number of a camera in the system (from 1 to 4).*

```
if(Event.GetSourceType() == "MACRO" && Event.GetAction() == "RUN")
{
```

```
var k;
//Switching the cameras to the Day mode by activating the *.1 detection zones
if(Event.GetSourceId() == "1")
{
 for(k=1; k<=2; k=k+1)
 {
  DoReactStr("CAM_ZONE", k + ".1", "ENABLE", "");
  DoReactStr("CAM_ZONE", k + ".2", "DISABLE", "");
  DoReactStr("CAM_ZONE", k + ".3", "DISABLE", "");
 }
}

//Switching the cameras to the Nigh mode by activating the *.2 detection zones
if(Event.GetSourceId() == "2")
{
 for(k = 1; k <= 2; k = k+1)
 {
  DoReactStr("CAM_ZONE", k + ".1", "DISABLE", "");
  DoReactStr("CAM_ZONE", k + ".2", "ENABLE", "");
  DoReactStr("CAM_ZONE", k + ".3", "DISABLE", "");
 }
}

//Switching the cameras to the Rain mode by activating the *.3 detection zones
if(Event.GetSourceId() == "3")
{
 for(k = 1; k <= 2; k = k+1)
 {
```

```
    DoReactStr("CAM_ZONE", k + ".1", "DISABLE", "");
    DoReactStr("CAM_ZONE", k + ".2", "DISABLE", "");
    DoReactStr("CAM_ZONE", k + ".3", "ENABLE", "");
   }
 }
}
```

1.2.2.3 *The GetSourceId method*

The GetSourceId method returns the identification number of the MsgObject or Event object.

Method call syntax

```
function  GetSourceId() : String
```

Method arguments

No arguments.

Usage examples

See the example for the GetSourceType method.

1.2.2.4 *The GetAction method*

The GetAction method returns the event received as an Event object or specified for a MsgObject object.

Method call syntax

```
function  GetAction() : String
```

Method arguments

No arguments.

Usage examples

See the example for the GetSourceType method.

1.2.2.5 *The GetParam method*

The GetParam method returns the value of the specified parameter of the MsgObject or Event object.

Method call syntax

```
function  GetParam(param: String) : String
```

Method arguments

param

Required argument. The name of the parameter of the object created using MsgObject (or of the Event object). Takes the following values: Type – String, range – available parameters for the objects of the specified type.

Usage examples

Problem. When an event from any camera occurs, check if this camera is set for sending color video signal. If a camera is set for sending black-and-white signal, switch it to color mode (parameter - color, value – true (1)).

```
if (Event.SourceType == "CAM")
{
 var msg = Event.Clone();
 if (msg.GetParam("color") == "0")
 {
  msg.SetParam("color", "1");
 }
 NotifyEvent(msg);
}
```

1.2.2.6 *The SetParam method*

The SetParam method assigns a value to the specified parameter of the MsgObject or Event object. It changes only the specified parameters, leaving other parameters intact.

Method call syntax

```
function  SetParam(param : String,  value : String)
```

Method arguments

param

Required argument. The name of the parameter of the object created using MsgObject (or of the Event object). Takes the following values: Type – String, range – available parameters for the objects of the specified type.

value

Required argument. The value to be assigned to the parameter specified in the param argument. Takes the following values: Type – String, range – depends on the parameter.

Usage examples

See the example for the GetParam method.

1.2.2.7 *The MsgToString method*

The MsgToString method transforms MsgObject objects (including the static Event object) into a String variable.

Method call syntax

```
function  MsgToString() : String
```

Method arguments

No arguments.

Usage examples

Problem. Send the messages about all events registered for Microphone 1, to a specified e-mail address.

Note. The Short Messages Service is supposed to be installed and working properly.

```
if (Event.SourceType == "OLXA_LINE" && Event.SourceId == "1")
{
  var msgstr = Event.MsgToString();
DoReactStr("MAIL_MESSAGE", "1", "SEND", "subject<Microphone 1>,body<" + msgstr + ">");
DoReactStr("MAIL_MESSAGE", "1", "SEND", "");
}
```

1.2.2.8 *The StringToMsg method*

The StringToMsg method transforms a String variable into an MsgObject object.

Method call syntax

```
StringToMsg(msg : String) : MsgObject
```

Method arguments

msg

Required argument. A String type variable to be transformed into an MsgObject object. Takes the following values: Type – String; range – character string that matches the syntax for MsgObject representation:

"objtype|id|action|param1<value1>,param2<value2>…", where

objtype – object type;

id – object identification number;

action – event or action for the object;

param1<value1>,param2<value2> - list of parameters and their values. Elements of the list are separated by commas with no white space. If no parameters need to be specified, an empty string is used after the vertical line (|), for example:

"CAM|1|MD_START|"

Usage examples

Problem. Upon an alarm in Sensor 1 or 3, start recording audio from Microphone 1. Upon an alarm in Sensor 2 or 4, start recording audio from Microphone 2.

```
if (Event.SourceType == "GRAY" && Event.Action == "ALARM")
{
 var audioid;
 if (Event.SourceId == "1" || Event.SourceId == "3")
 {
  audioid = "1";
 }
 if (Event.SourceId == "2" || Event.SourceId == "4")
 {
  audioid = "2";
 }
 var str = "OLXA_LINE|"+audioid+"|ARM|";
 var msg = CreateMsg();
 msg.StringToMsg(str);
 NotifyEvent(msg);
}
```

1.2.2.9 *The StringToParams method*

The StringToParams method transforms a String variable into the list of parameters and overwrites the existing parameter list of the MsgObject object.

Method call syntax

```
StringToParams(String params)
```

Method arguments

params

Required argument. A String type variable to be transformed into a list of parameters for the MsgObject object. Takes the following values: String variables matching the syntax for MsgObject parameter list representation:

“param1<value1>,param2<value2>…”, where

param1<value1>,param2<value2> - list of parameters and their values. Elements of the list are separated by commas with no white space. If no parameters need to be specified, an empty string is used after the vertical line (|), for example:

“CAM|1|MD_START|”

Additional information

Usage examples

Problem. Upon registration the connection (the “attach” event), set new values to the following parameters of any camera: the number of the PTZ device (telemetry_id) and the number of the microphone for synchronous recoeding (audio_id). The values should be equal to the corresponding camera numbers plus 1.

```
if (Event.SourceType == "CAM" && Event.Action == "ATTACH")
{
  var i;
  for (i=1;i<=4;i=i+1)
 var msg = Event.Clone();
 var str = "telemetry_id<" + (i+1) +">,audio_id<" + (i+1) + ">";
 msg.StringToParams(str);
 NotifyEvent(msg);
}
```

1.2.2.10 *The Clone method*

The Clone method creates a copy of an MsgObject or Event object.

Method call syntax

```
Clone() : MsgObject
```

Method arguments

No arguments.

Usage examples

Problem. When Relay №1 closes, start video recording from Camera 1 and close Relay №2. When Relay №1 opens, start video recording from Camera 2 and open Relay №2.

```
if (Event.SourceType == "GRELE" && Event.SourceId == "1")
{
 var msgevent = Event.Clone();
 if(Event.Action == "ON")
 {
  msgevent.SourceId = "2";
  DoReact(msgevent);
  DoReactStr("CAM","1","REC","");
  DoReactStr("GRELE","2","ON","");

 }
 if(Event.Action == "OFF")
 {
  msgevent.SourceId = "2";
  DoReact(msgevent);
  DoReactStr("CAM","2","REC","");
  DoReactStr("GRELE","2","OFF","");

 }
}
```

1.2.2.11 *The SourceType property*

The SourceType property stores the system type of the MsgObject or Event object.

Property call syntax

```
SourceType : String
```

Usage examples

Problem. When Relay 1 closes (for example, the button connected to the relay is pressed), print the frames from Cameras 1 and 2.

```
if (Event.SourceType == "GRELE" && Event.SourceId == "1" && Event.Action == "ON")
{
```

```
 //activating the Camera 1 window
 DoReactStr("MONITOR","1","ACTIVATE_CAM", "cam<1>");
 //printing the frame from Camera 1
 DoReactStr("MONITOR","1","KEY_PRESSED","key<PRINT>");
 //activating the Camera 2 window
 DoReactStr("MONITOR","1","ACTIVATE_CAM", "cam<2>");
 //printing the frame from Camera 2
 DoReactStr("MONITOR","1","KEY_PRESSED","key<PRINT>");
}
```

1.2.2.12 *The SourceId property*

The SourceType property stores the identification number of the MsgObject or Event object.

Property call syntax

SourceId : String

Usage examples

See the example for the SourceType property.

1.2.2.13 *The Action property*

The Action property stores the action or event of the MsgObject or Event object.

Property call syntax

SourceId : String

Usage examples

See the example for the SourceType property.

1.2.2.14 *The GetObjectIds method*

GetObjectIds method is responsible for receiving identifiers from all the objects of a specified type.

Method call syntax:

function GetObjectIds(objectType : String)

A line is replied :

CAM||COUNT|id.3<5>,id.count<4>,id.0<2>,id.1<3>,id.2<4>

where id.count<> – number of ID objects,

id.[count]<> – ID object.

Method's arguments:

objectType –required argument. Set the type of the system object, for which the value of the given parameter should be given back ("CAM","GRAY","GRABBER" e.t.c.).Accepted values: type String, range is restricted by object types registered in the system.

Example. All the cameras should be armed upon the start of MacrosN°1.

```
if (Event.SourceType == "MACRO" && Event.SourceId && Event.Action == "RUN")
{
	var msg = CreateMsg();
	msg.StringToMsg(GetObjectIds("CAM"));
	var objCount = msg.GetParam("id.count");
	var i;
	for(i = 0; i < objCount; i++)
	{
		DoReactStr("CAM", msg.GetParam("id." + i), "ARM", "");
	}
}
```

1.2.2.15 *The GetObjectParams method*

GetObjectParams method is designed for getting the object's parameters.

Method call syntax:

function GetObjectParams(objectType : String, objectId : String)

Method arguments:

objectType – required argument. Set the type of the system object ("CAM", "GRAY", "GRABBER" e.t.c.), for which the type of a parent object should be given back. Accepted values: type String, range is restricted by object types registered in the system.

objectId – object's identifier. Accepted values: String type.

Example. It is necessary to check the color control of camera N°2 upon the start of MacrosN°1. If camera 2 is a color one, set it to recording.

```
if (Event.SourceType == "MACRO" && Event.SourceId && Event.Action == "RUN")
{
	var msg = CreateMsg();
```

```
        msg.StringToMsg(GetObjectParams("CAM", "2"));
        if(msg.GetParam("color") == "1")
        {
            DoReactStr("CAM", "2", "REC", "");
        }
}
```

1.3 Programming tools

1.3.1 The Script object

The Script object is designed for initializing and setting the parameters of JavaScript scripts in the Intellect system.

Figure 1.3-1 shows the Script object settings panel.

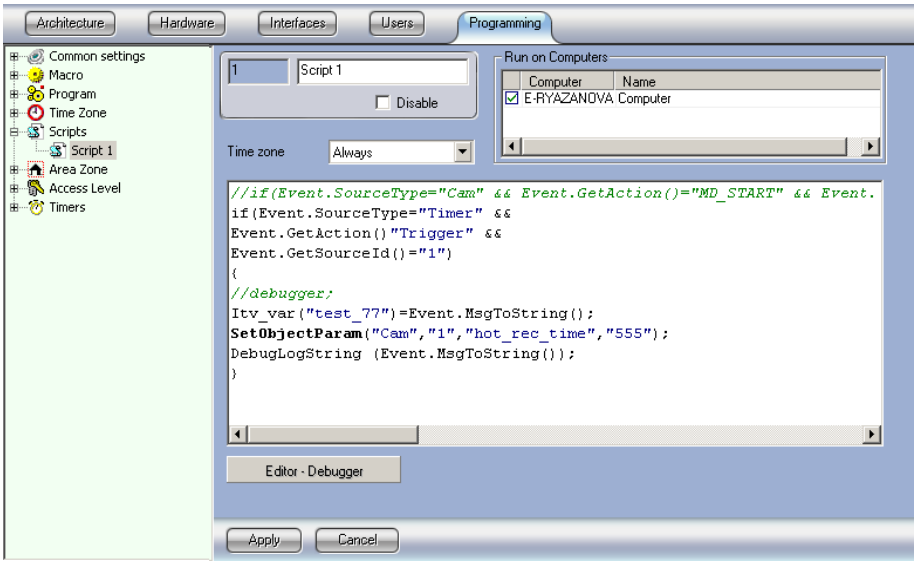

Figure 1.3-1. The Script object settings panel

The settings panel allows choosing the time zone and the computers (kernels) for running the script.

It also includes the button for launching the Editor-Debugger utility and the text window for viewing the script text created in the utility. The scripts can be edited in the Editor-Debugger utility or can be edited directly in the Script object settings panel.

1.3.2 The Editor-Debugger utility

The Editor-Debugger utility is designed for creating, debugging and editing scripts in the Intellect software package.

Figure 1.3-2 shows the Editor-Debugger dialog window.

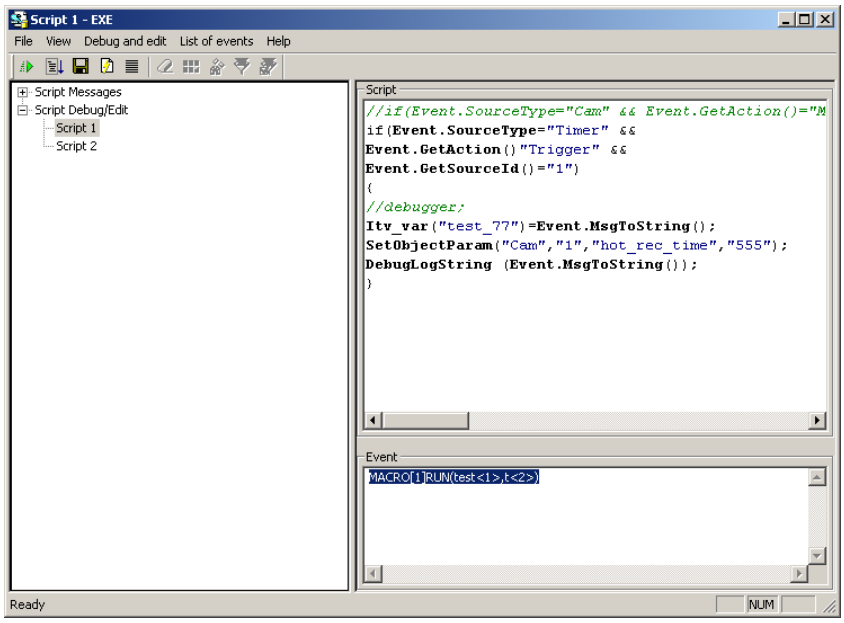

Figure 1.3-2. The Editor-Debugger dialog window

The Editor-Debugger utility contains the built-in text editor and the debugger window.

To help with writing correct codes, the text editor automatically highlights objects, methods and properties in different colors.

The debugger window allows viewing the information about all events registered by the system. You may filter the events to be shown in the debugger window. A separate debugger window is created for each Script object, which allows each script to be debugged individually using the filters.

To debug a script, the utility can generate test events which will not be registered by the system.

Scripts can be saved as Script objects or as text files on the hard drive.

1.3.3 The debugger window

The debugger window is used for viewing all events registered by the system.

To open the debugger window, use the Debugger Window item in the Run menu in the main control panel. The debugger window opens in the lower part of the screen (Figure 1.3-3).

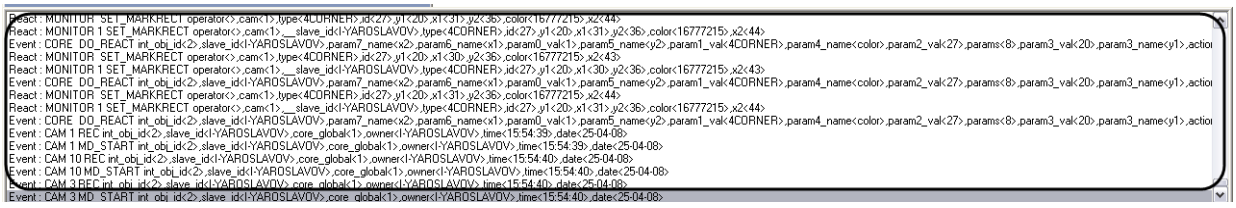

Figure 1.3-3. The debugger window in the Intellect software package

By default, the menu item for opening the debugger window is not available. To activate the opening of the debugger window, use the Tweaki.exe utility (see the Intellect Software Package – Administrator's Guide document).

1.4 Creating, saving and deleting scripts

1.4.1 Creating a script

To create a JavaScript script in Intellect, the Editor-Debugger utility is used.

To start the utility, click the Editor-Debugger button in the Script object settings panel.

To create a script, do the following:

1. In the Programming tab of the System Settings window, create a Script object. Enter the identification number and the name for the script.

2. Select the time limits for running the script in the Time Zone field (for example, the Always zone).

Note. By default, the Never time zone is selected.

3. In the Computers field, select the computers (kernels) where the script should run.

Note. By default, the script will run on all computers. The list shows only the computers registered in the Hardware tab of the System Settings window.

4. Click the Editor-Debugger button in the bottom of the Script settings panel to open the Editor-Debugger utility.

Note. The scripts can be created, edited and saved in the Editor-Debugger utility only. The Script object settings panel displays only the read-only text of the script.

5. In the Editor-Debugger utility window, open the Script Debug/Edit list and select the Script object to be edited (for example, Script 2 in Figure 1.4-1).

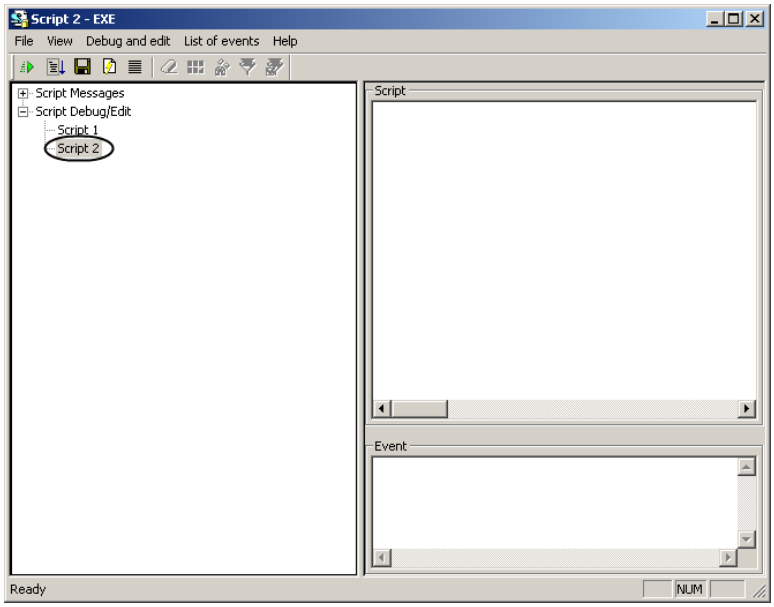

Figure 1.4-1. Starting script editing using the Editor-Debugger utility

6. Enter the text of the script in JavaScript programming language into the Script field.

7. Run the script using a test event. To create a test event, select the Edit test event in the Debug and edit menu. The Test message window will open allowing to set the test event parameters (Figure 1.4-2).

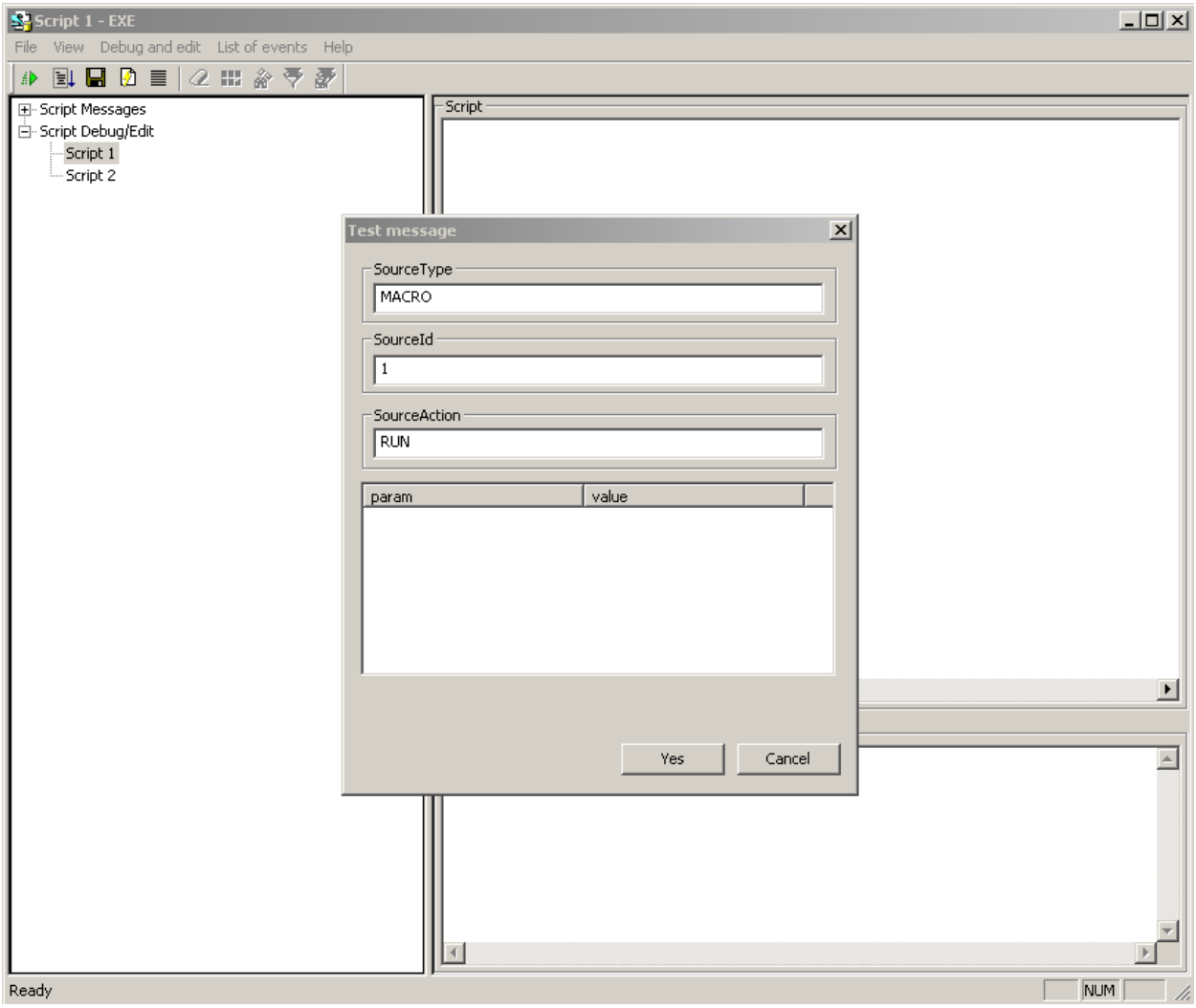

Figure 1.4-2. The test event editing window with fulfilled fields

To run the script using the test event, select Test run in the Debug and Edit menu.

1. Check the script syntax using the interpreter which is built-in into the Editor-Debugger utility. The verification results showing the error and its location will be displayed in the debugger window of this script in the Script messages list. Correct the script to eliminate the error and repeat the verification.

Note. See the detailed information about using test events for script debugging in the Script Debugging section.

2. After debugging the script using the Editor-Debugger utility, run it with a real event. Check the result. If the result is incorrect, change it and run again.

Script creation is considered complete if it runs correctly.

1.4.2 Saving a script

The Editor-Debugger utility provides two options for saving scripts – in a Script object, or in a text file on the hard drive.

To save the script in the Script object, in the Debug and edit menu, select Save, or in the File menu, select Save in the database.

Note. The script is automatically saved in the corresponding Script object upon closing the Editor-Debugger utility.

To save the script in the file, in the File menu, select Save on disk. To open a script saved in a file in the Editor-Debugger utility, in the File menu, select Open from disk.

1.4.3 Deleting a script

To delete a script created in the Intellect system, delete the corresponding Script object in the Programming tab.

1.5 Creating your first script

As an example of using JavaScript in Intellect, try to create a script for the following tasks: when Macro 1 starts, set the value 10 to the Hot Recording parameter for Cameras № 1 – 4 and output the “Hello world” message to the debugger window of the Editor-Debugger utility.

To create and run this script, do the following:

1. In the Hardware tab of the System Settings window, create four Camera objects with identification numbers 1, 2, 3 and 4, if they have not been created before.
2. In the Programming tab, create a Macro object with identification number 1. Fill in the Events table with the following values: in the Type column, enter “Macro”, in the Number column, enter 1, in the Event column, enter “Action executed”.
3. Create a Script object in the Programming tab. Enter the identification number 1 and the name “Script 1”.
4. In the Script 1 object settings panel, select Always in the Time zone list.
5. Click the Editor-Debugger button at the bottom of the Script 1 settings panel. The Editor-Debugger window will open.
6. In the Editor-Debugger window, open the Script Debug/Edit list and select the Script 2 object (Figure 1.5-1).

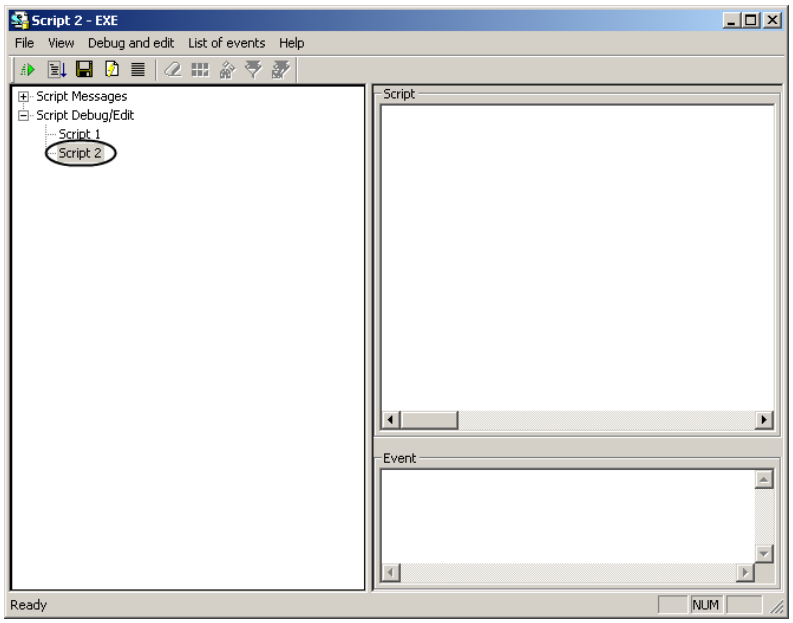

Figure 1.5-1. Selecting the Script 1 object in the Editor-Debugger window

7. Enter the following in the Script field:

```
if (Event.SourceType == "MACRO" && Event.SourceId == "1" && Event.Action == "RUN")
{
 var ;
 for(i=1; i<=4; i=i+1)
 {
  SetObjectParam("CAM",i,"hot_rec_time","10");
 }
 DebugLogString ("Hello world");
}
```

8. In the File menu, select Save to database to save the script.

9. Create a test event to run the script in debug mode – “MACRO|1|RUN|”. To achieve this, in the Debug and edit menu, select Edit test event; the Test message window will open. Fill in the fields in the Test message window as shown in Figure 1.5-2.

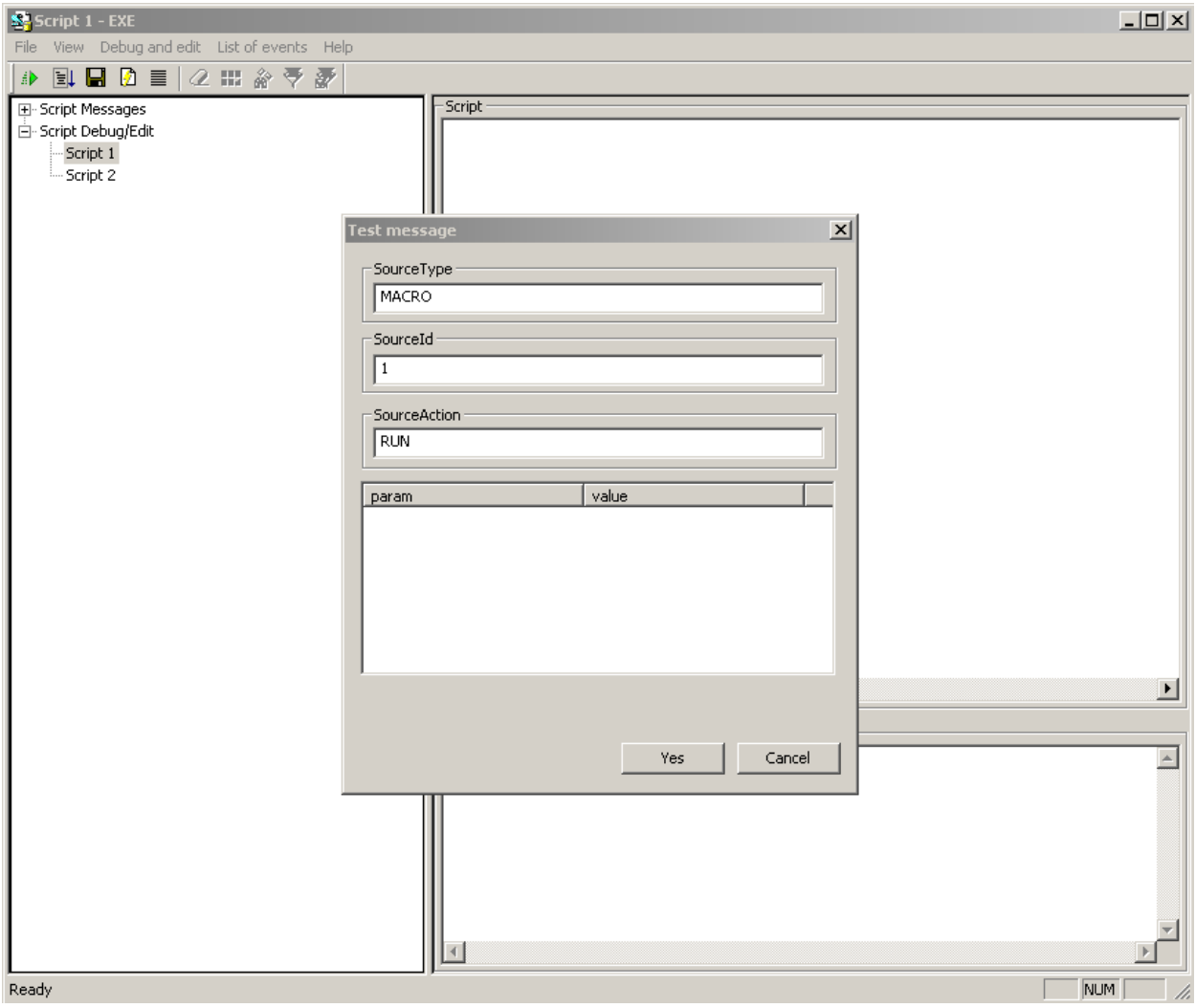

Figure 1.5-2. The Test message window with the "Run Macro 1" test event

10. To run the script with the test event, select Test run in the Debug and edit menu.

11. Open the Script Messages list and select Script 1. The debugger window will open at the right side.

12. In the debugger window, find the "Process Event:MACRO|1|RUN|" line and the following error message: "Src identifier missing: Microsoft JScript compilation error Line:2 Char:6 Error:0 Scode:800a03f2" (Figure 1.5-3).

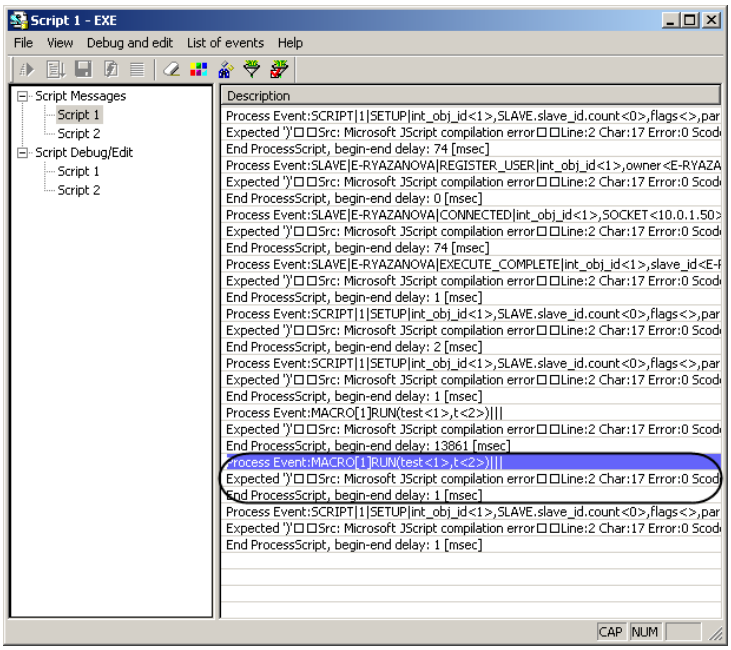

Figure 1.5-3. Script execution error message

The message says that the variable declaration procedure (var) has been called, but no variable has been declared. This is a critical error in JavaScript, thus the script has not been executed.

13. Change the text of the script (see the “var i;” line).

```
if (Event.SourceType == "MACRO" && Event.SourceId && Event.Action == "RUN")
{
 var i;
 for(i=1; i<=4; i=i+1)
 {
  SetObjectParam("CAM",i,"hot_rec_time","10");
 }
 DebugLogString ("Hello world");
}
```

14. In the File menu, select Save to database to save the script.
15. Repeat steps 10 and 11.
16. In the debugger window, find the “Process Event:MACRO|1|RUN|” line and the “DebugLogString:Hello world” and “Script first run OK” messages. The “Script first run OK” means that the script runs correctly in the debug mode (Figure 1.5-4).

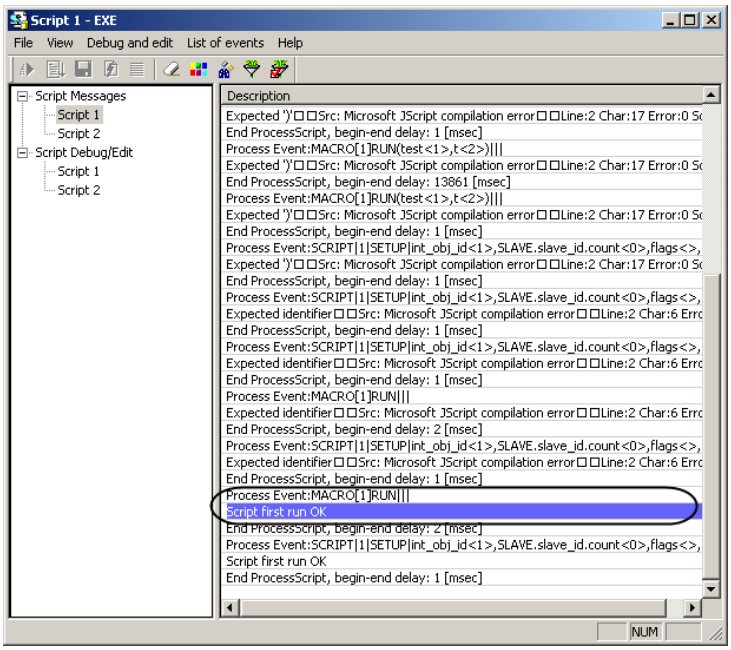

Figure 1.5-4. Message about correct execution of the script

17. Close the Editor-Debugger utility.

18. The text of the created script will be shown in the Script field of the Script 1 object settings panel. Click the Apply button in the Script 1 settings panel to activate the script.

19. Select Macro 1 in the Run menu of the main control panel.

20. In the debugger window of the Intellect system, check that the macro and the script have run successfully (Figure 10).

Figure 1.5-5. Debugger window showing macro execution

21. Check the accuracy of the script result. The Hot recording field in the Camera 1 to Camera 4 object settings panels should read “10” (Figure 1.5-6).

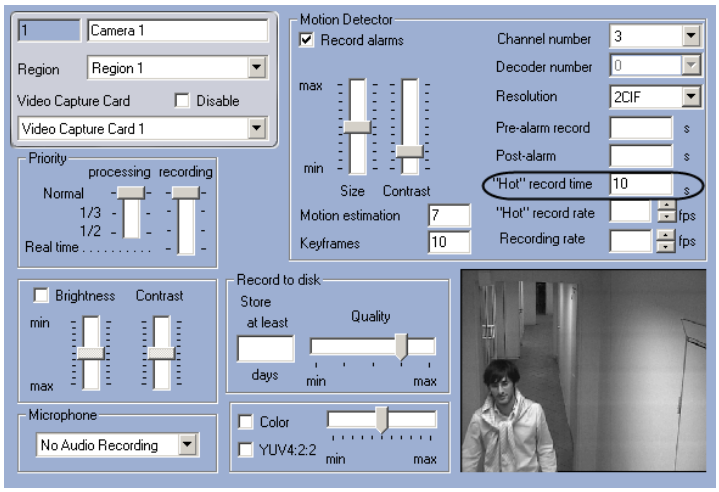

Figure 1.5-6. The value in the Hot recording field after running the script

Note. The Hot recording field in the Camera settings panel is empty by default.

Script creation and debugging is now complete.

1.6 Script debugging

1.6.1 Script debugging features

The Editor-Debugger utility allows debugging scripts using the built-in tools for checking script syntax, script interpreting and script execution with test events generated by the utility. The messages about the debugging results are displayed in the corresponding debugger windows.

The Editor-Debugger utility provides the following debugging functionality:

1. A separate debugger window is assigned to each Script object, where the test and system events, error messages, success messages and user information messages are displayed. The messages in the debugger windows can be filtered.
2. Special Information window debugger windows are available for displaying the messages related to the script being debugged.
3. Test events generated by the Editor-Debugger utility, which are not registered by the Intellect system, are used for checking script accuracy.
4. Third-party debugger programs can be used for step-by-step execution, viewing script variables during execution, and other functionality.

1.6.2 Creating and using test events

1.6.2.1 *Creating test events*

The Editor-Debugger utility is capable of generating test events chosen by the user to help debug the scripts. Test events are not registered by the video surveillance system, i. e. they are not listed in the events log and not saved to the database.

No more than one test event can be created for each script.

To create a test event, do the following:

1. In the Debug and edit menu, select Edit test event, or click the  button in the toolbar.
2. The Test message window will open (Figure 1.6-1). This window is used for entering the test event parameters.

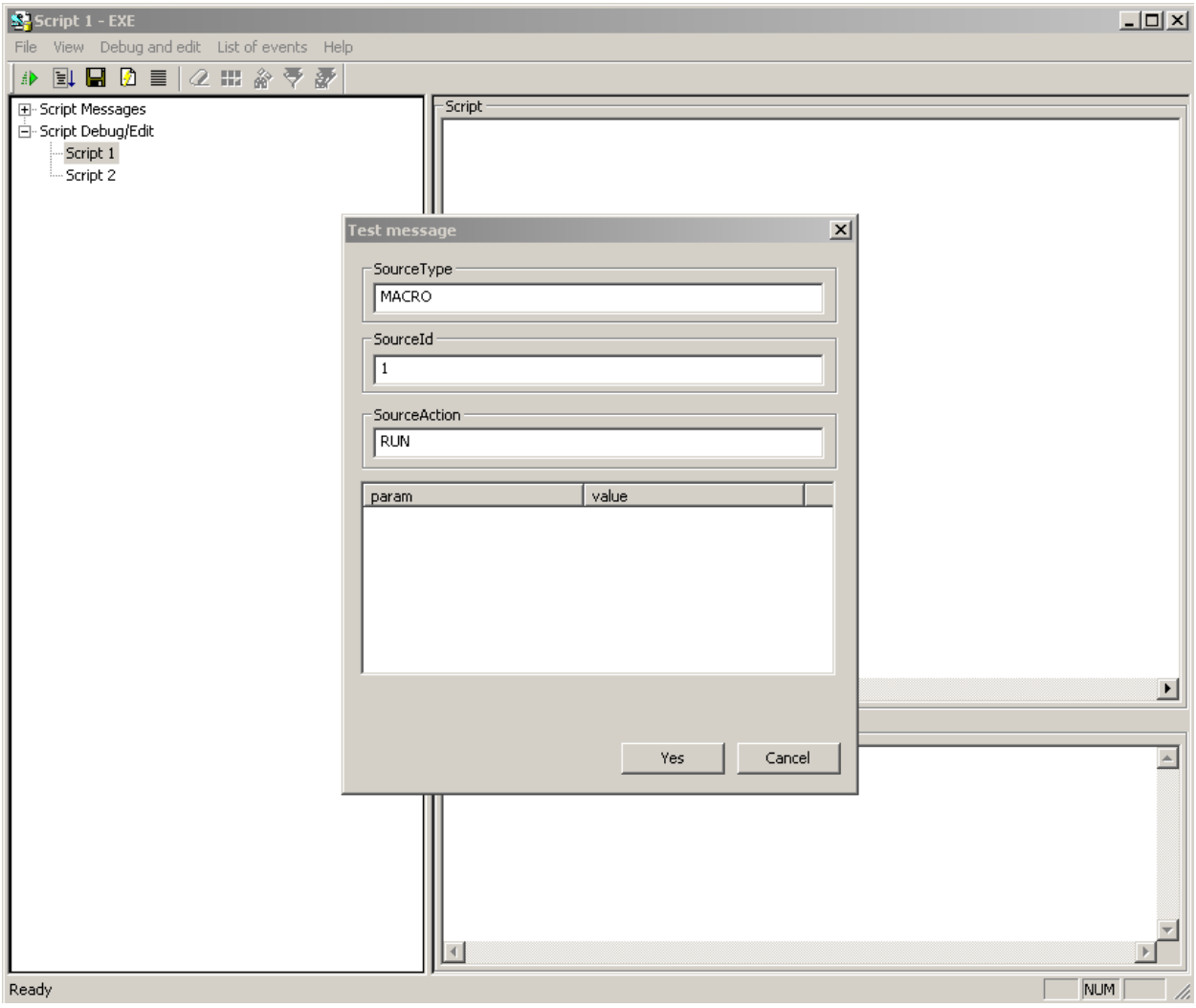

Figure 1.6-1. The Test message window

3. Enter the following information in the fields of the Test message window:
    a. SourceType – object type;
    b. SourceId – object identification number;
    c. SourceAction – the event generated by the specified object;
    d. param – additional event parameters;
    e. value – values of the additional parameters.
4. Click the OK button.

The test event is now created.

The created test event will be displayed in the Event field in a special string format.

Figure 1.6-2 shows the test event example: “Arm Camera 111”.

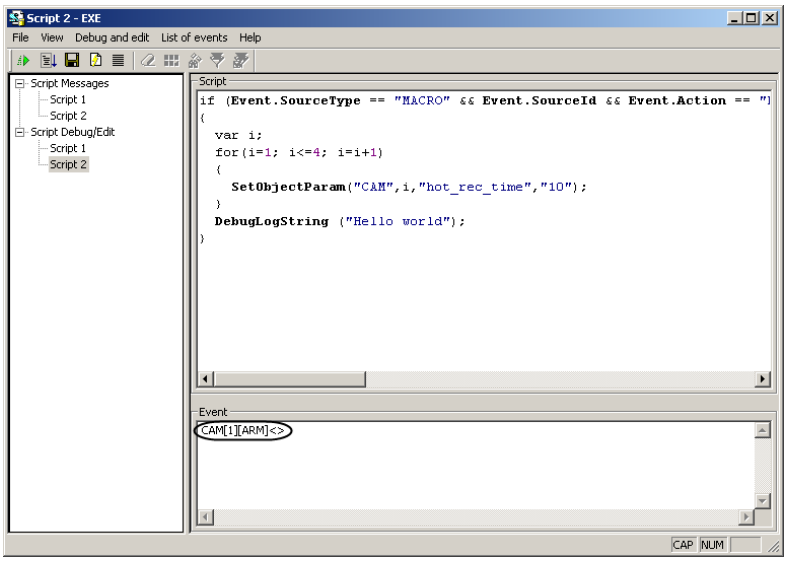

Figure 1.6-2. Test event example

1.6.2.2 *Running the script with a test event*

To run the script with a test event, do one of the following:

1. Click the Test run ( ) button in the toolbar.
2. In the Debug and edit menu, select Test run.
3. In the Debug and edit menu, select Test run in third-party debugger.

When the Test run in third-party debugger option is selected, the third-party debugger starts to run the test (see details in the Using third-party debuggers section).

The results of the verification and execution of the script are displayed in the corresponding debugger window of the Editor-Debugger utility.

1.6.3 Using debugger windows of the Editor-Debugger utility

1.6.3.1 *Debugger window types: Script Messages and Thread Information*

Debugger windows display messages about system and test events, errors and successful script execution, as well as user information messages.

A separate debugger window is assigned to each script in the Editor-Debugger utility.

There are two types of debugger windows: Script Messages and Thread Information.

The names of Script Messages windows are listed in the Script Messages list. The names of the windows match the names of the corresponding Script objects. These windows display all system messages related to all scripts in the Intellect system (Figure 1.6-3).

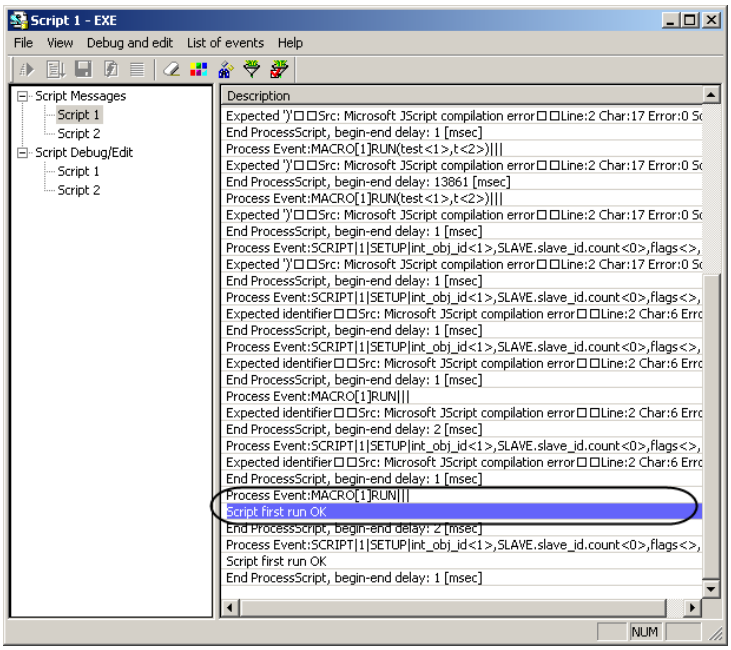

Figure 1.6-3. An example of a Script Messages window

The Thread Information windows open directly from the script editing windows (named in the Script Debug/Edit list). To open the Thread Information window from an active script editing window, select Summary information in the Debug and edit menu, or click the  button in the toolbar. The Thread Information windows display only the events related to the current script being debugged (Figure 1.6-4).

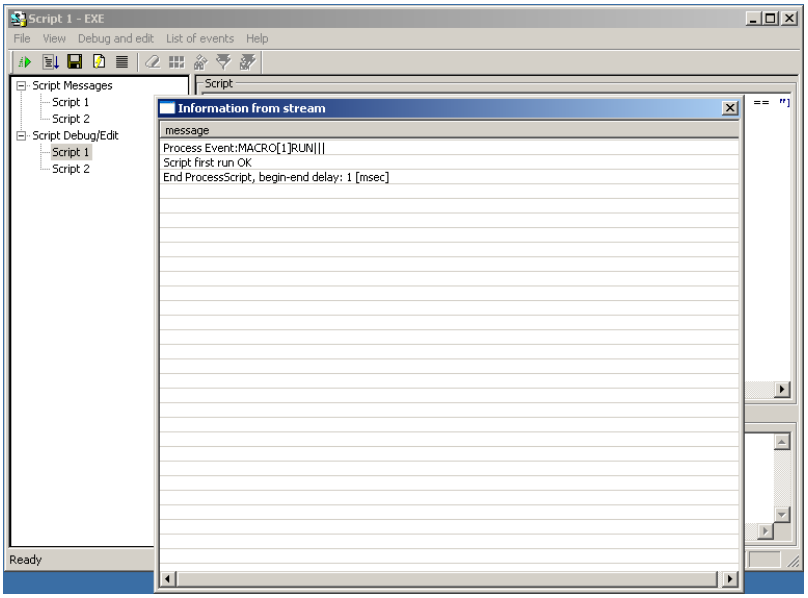

Figure 1.6-4. The Thread Information window

Both types of debugging windows are used in the same way.

1.6.3.2 *Displaying messages about starting, verifying and executing scripts in the debugger windows*

The messages in the debugger window track the stages of starting, verification and execution of scripts.

When the event occurs that triggers the script, the following message is displayed in the debugger window: “Process Event: <script triggering event>”; for example, if the script starts upon Macro 1 execution, the line reads “Process Event:MACRO|1|RUN|”.

Script syntax is checked before execution. In case of syntax errors, related error messages will be displayed in the debugger window. Figure 1.6-5 shows an example of a syntax error message.

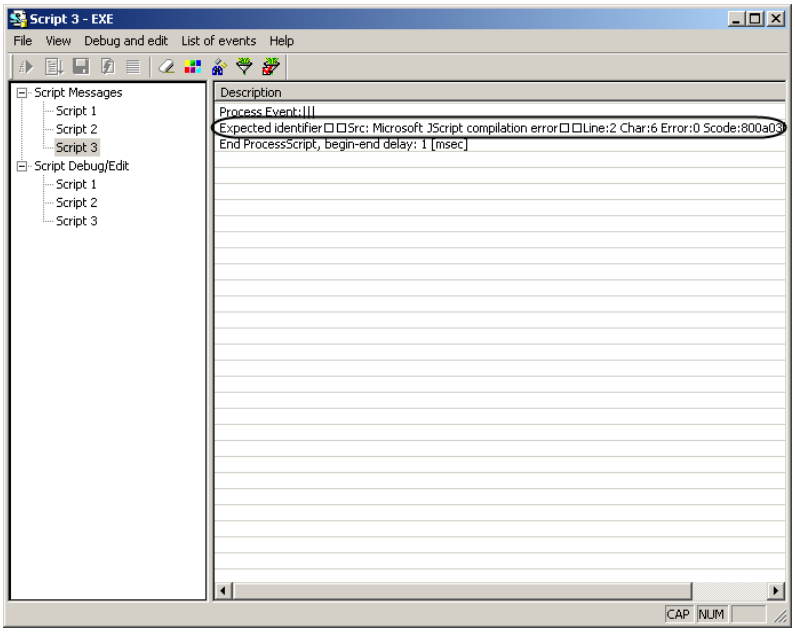

Figure 1.6-5. Syntax error message

Right-click the message to view its complete text. The **Information** window will open containing the full text of the error message (Figure 1.6-6).

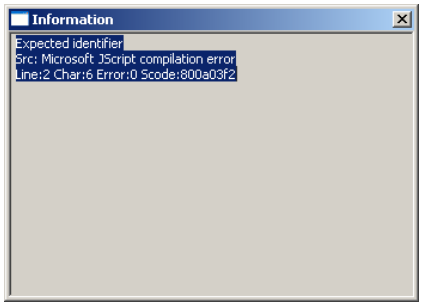

Figure 1.6-6. Information window with the error message

The message contains the following information:

1. Error description;
2. Error type (for example, “Src: Microsoft Jscript compilation error”);
3. Error location in the script text (line number and character number in the line);
4. Error code (“Scode”).

In case no errors were detected, the following message will be displayed in the debugger window: «Script first run OK». Then, the script will run.

Script runtime errors are also displayed in the debugger window.

In case of successful execution of the script, the following message will be displayed: “End ProcessScript, begin-end delay: <script execution time>; for example, “End ProcessScript, begin-end delay: 13 [msec]”.

1.6.4 Using third-party debugger programs

The Intellect software package allows using third-party debuggers for processing JavaScript scripts. These programs may have the functionality that is not included in the Editor-Debugger utility, for example, step-by-step script execution, watching script variables during script execution, etc.

Note. We do not recommend using third-party debuggers, since they do not provide full compatibility with the Intellect software, and may lead to failure of the Intellect software.

We strongly recommend introducing the breakpoint in the script when using third-party debuggers. To insert the breakpoint, add the following line to the script: “debugger;”. The script execution will pause at this line, and the debugger will start.

When a third-party debugger is used, scripts can be started with test events only.

To start the script using a third-party debugger, do the following:

1. Create the script and insert the “debugger;” line in it.
2. Create a test event for the script.
3. In the Debug and edit menu, select Test run in third-party debugger.
4. The Just-In-Time Debugging window will open. Select one of the debuggers installed on the computer (Figure 1.6-7).

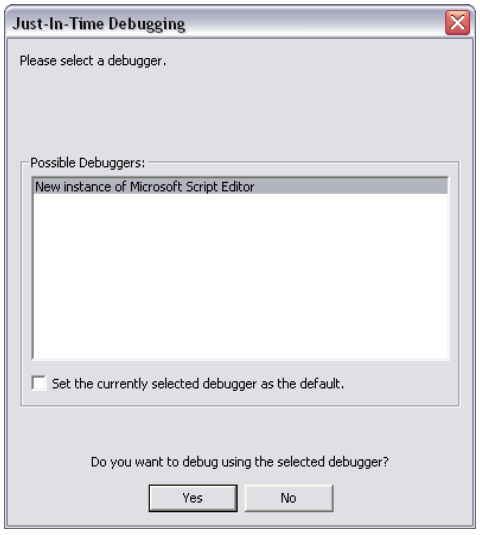

Figure 1.6-7. Selecting the third-party debugger

1. Click **Yes** to confirm the selection.

In case of a successful syntax check (no errors found before the “**debugger;**” line), the third-party debugger will start. The script will pause at the breakpoint.

Example.A script with the use of the breakpoint after starting macros №1.

```
if (Event.SourceType==”MACRO” && Event.SourceId==”1” && Event.Action == “RUN”); \\starting macros №1

{

debugger; \\breakpoint

DebugLogString (“Hello world”);

}
```

2 Appendix 1. Description of the Editor-Debugger utility

2.1 The purpose of the Editor-Debugger utility

The Editor-Debugger utility is designed for creating, debugging and editing scripts in the Intellect software package.

The Editor-Debugger utility provides the following functionality:

1. Creating and editing scripts using the built-in text editor;
2. Debugging scripts using the built-in debugger window;
3. Filtering the information to be displayed in the debugger window;
4. Creating and using test events for debugging;
5. Saving scripts to the hard drive;
6. Opening scripts from the hard drive.

2.2 The interface of the Editor-Debugger utility

2.2.1 The Editor-Debugger window

The Editor-Debugger window contains the main menu, the toolbar (Figure 2.2-1, 1), the object list (Figure 2.2-1, 2) and the viewing/editing panel (Figure 2.2-1, 3).

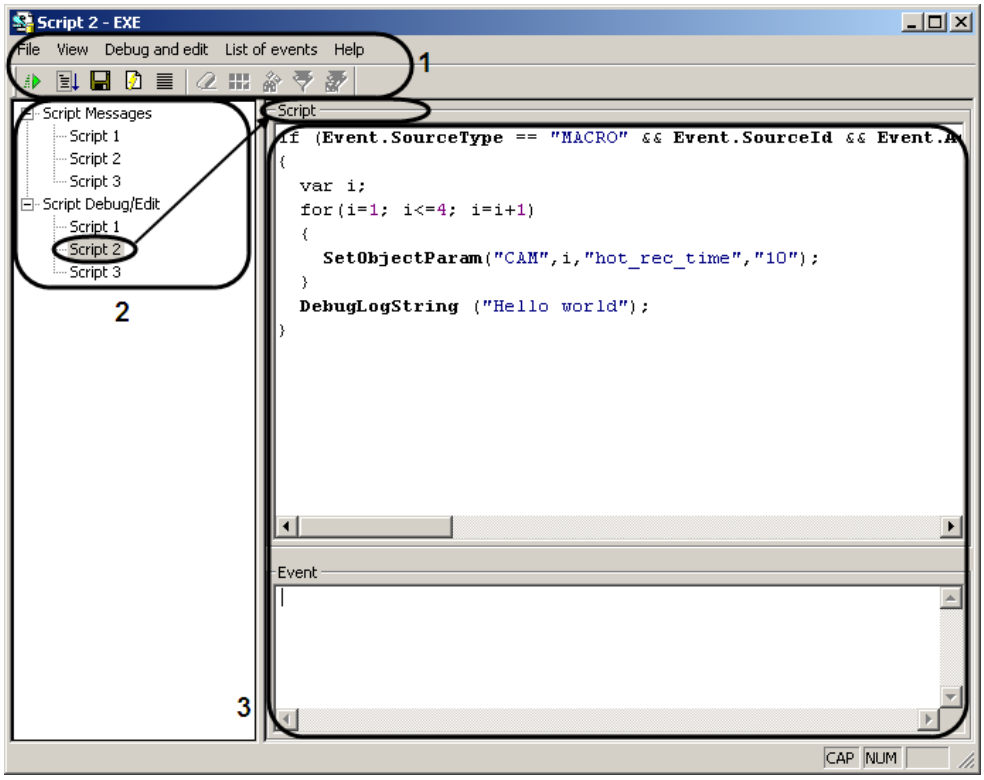

Figure 2.2-1. The Editor-Debugger window

2.2.2 The Script Debug/Edit tab

2.2.2.1 *Description of the Script Debug/Edit tab*

The Script Debug/Edit tab is used for editing scripts and creating test events.

Figure 2.2-2 shows the Script Debug/Edit panel.

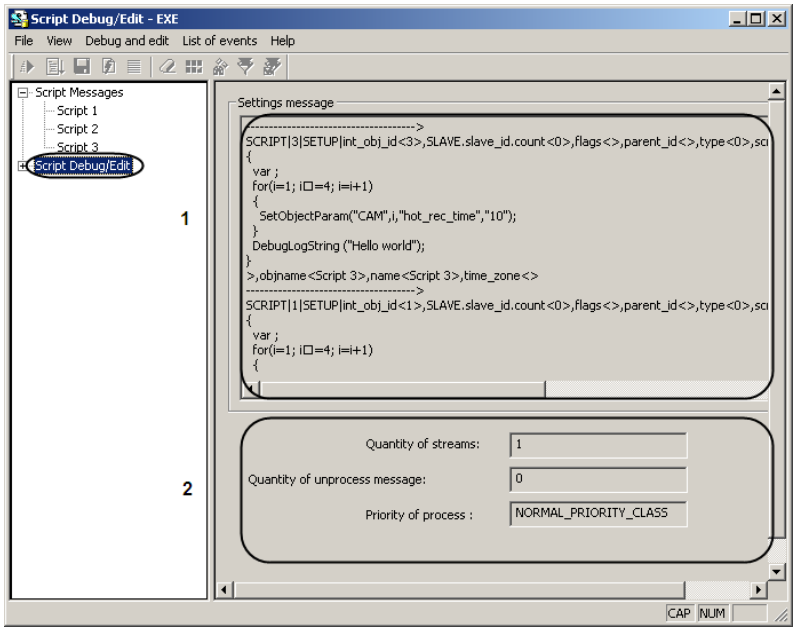

Figure 2.2-2. The Script Debug/Edit tab

Table 2.2-1 shows the description of the elements in the Script Debug/Edit panel.

Table 2.2-1

№	Field name	Field type	Description	Units	Default value	Value range
1	Settings message	Auto	Script object initialization information	Latin, Cyrillic and special symbols	-	-
2	Additional information	Auto	Additional information about scripts	Latin, Cyrillic and special symbols	-	-

2.2.2.2 *The Script object panel in the Script Debug/Edit tab*

The Script objects in the Script Debug/Edit tab are used for creating and editing scripts and test events.

Figure 2.2-3 shows the Script object panel.

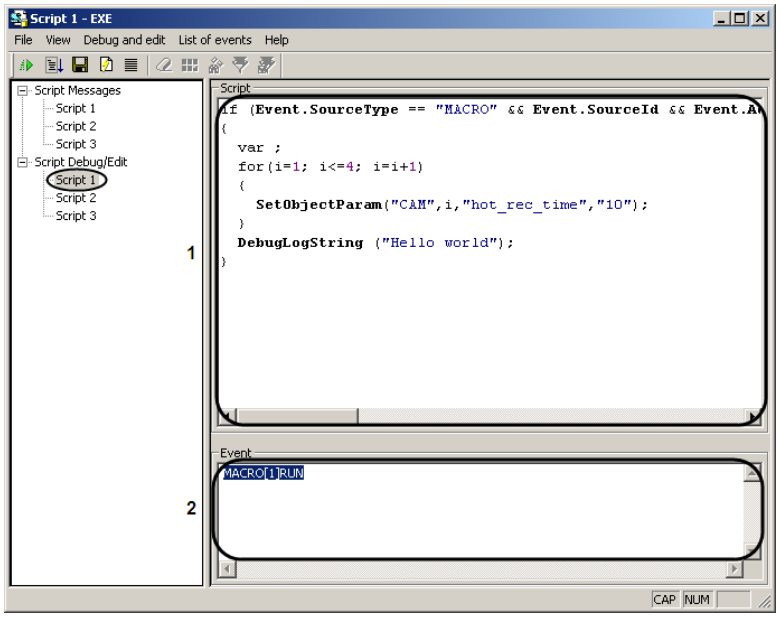

Figure 2.2-3. The Script object in the Script Debug/Edit tab

Table 2.2-2 describes the elements in the Script object.

Table 2.2-2

№	Field name	Field type	Description	Units	Default value	Value range
1	Script	Text field	The text of the script	Latin, Cyrillic and special symbols	Empty	Unlimited number of characters
2	Event	Text field	Event description in the line format	Latin, Cyrillic and special symbols	Empty	Unlimited number of characters

2.2.3 The Script Messages tab

2.2.3.1 *Description of the Script Messages tab*

The Script Messages tab is used to display the debugger windows for the scripts.

Figure 2.2-4 shows the Script Messages panel.

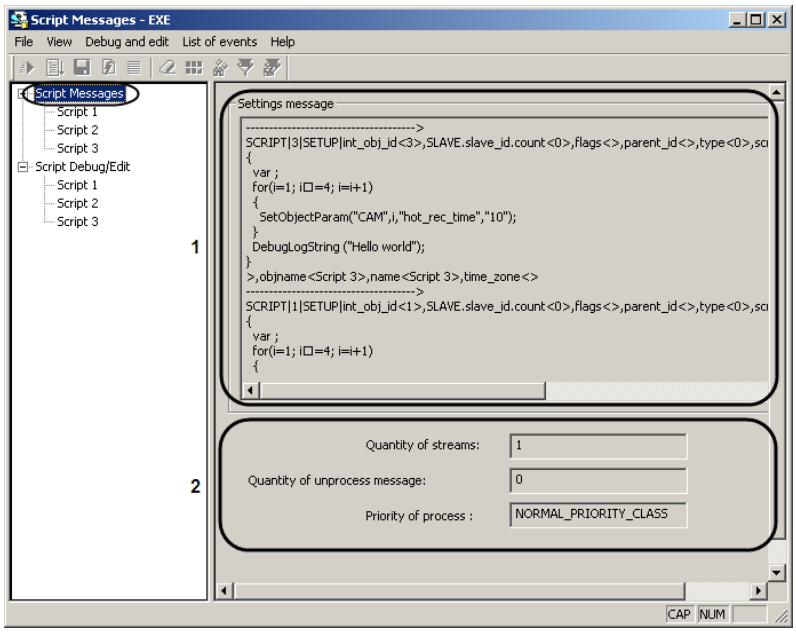

Figure 2.2-4. The Script Messages tab

Table 2.2-3 shows the description of the elements in the Script Messages panel.

Table 2.2-3

№	Field name	Field type	Description	Units	Default value	Value range
1	Settings message	Auto	Script object initialization information	Latin, Cyrillic and special symbols	-	-
2	Additional information	Auto	Additional information about scripts	Latin, Cyrillic and special symbols	-	-

2.2.3.2 *The Script object panel in the Script Messages tab*

The Script panels in the Script Messages tab are used to display system, test and user events related to the scripts created in Intellect.

Figure 23 shows the Script object panel.

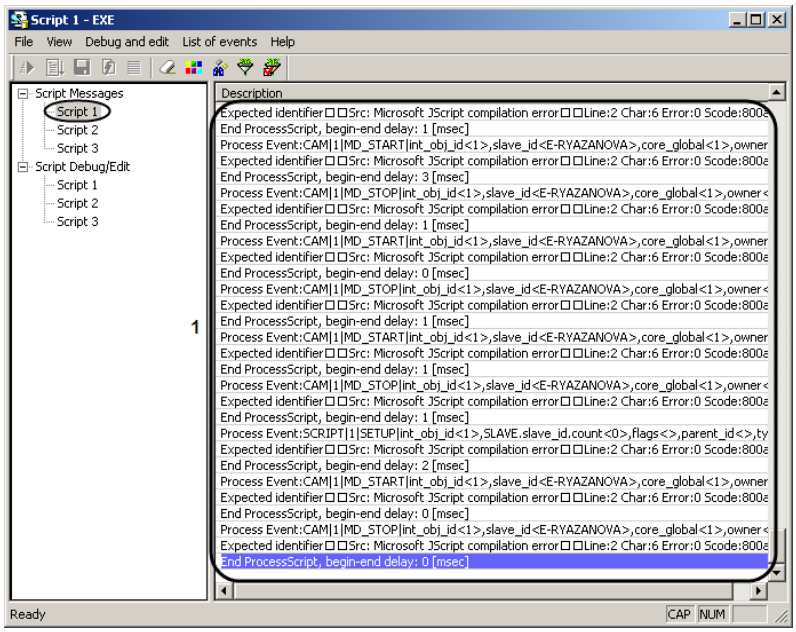

Figure 2.2-5. The Script object in the Script Messages tab

Table 2.2-4 describes the elements in the Script object.

Table 2.2-4

№	Field name	Field type	Description	Units	Default value	Value range
1	Description	Auto	Information about the events occurring in the system	Latin, Cyrillic and special symbols	Unspecified	-

2.2.4 Main menu

2.2.4.1 *Description of the main menu*

The Editor-Debugger main menu is used to call editing, debugging and other commands. The commands are grouped into functional menus: File, View, Debug and edit, Message list and Help.

Table 2.2-5 describes the elements of the main menu.

Table 2.2-5

№	Element name	Element type	Description	Units	Default value	Value range
1	File	Drop-down list of items	Commands for opening and saving scripts and closing the utility	-	-	
2	View	Drop-down list of items	Commands for displaying the toolbar and the status bar in the utility window	-	-	-
3	Debug and Edit	Drop-down list of items	Commands for script debugging	-	-	-

4	Message list	Drop-down list of items	Commands for changing the message display parameters in the debugger windows	-	-	-
5	Help	Drop-down list of items	The About command showing general information about the Editor-Debugger utility.	-	-	-

2.2.4.2 *The elements in the File menu*

The File menu is used to open and save scripts and to close the utility.

Table 2.2-6 describes the elements of the **File** menu.

Table 2.2-6

No.	Element name	Element type	Description	Units	Default value	Value range
1	Save to database	Item	Saves the script in the object	-	-	-
2	Save to disk	Item	Saves the script into a text file on the hard drive			
3	Open from disk	Item	Opens the script file	-	-	-
4	Exit	Item	Shuts down the utility and closes the window	-	-	-

2.2.4.3 *The elements in the View menu*

The View menu contains commands for showing and hiding the toolbar and the status bar.

Table 2.2-7 describes the elements of the View menu.

Table 2.2-7

№	Element name	Element type	Description	Units	Default value	Value range
1	Toolbar	Checkbox	Shows or hides the toolbar	Boolean	Checked	Check – show toolbar Uncheck – hide toolbar
2	Status bar	Checkbox	Shows or hides the status bar	Boolean	Checked	Check – show status bar Uncheck – hide status bar

2.2.4.4 *The elements of the Debug and edit menu*

The Debug and edit menu contains the commands for debugging scripts.

Table 2.2-8 describes the elements of the Debug and edit menu.

Table 2.2-8

№	Element name	Element type	Description	Units	Default value	Value range
1	Test run	Item	Runs the script with the test event	-	-	
2	Test run in third-party debugger	Item	Runs the script using the third-party debugger	-	-	-
3	Save	Item	Saves the script to the Script object	-	-	-
4	Edit test event	Item	Opens the window for editing test events	-	-	-
5	Summary information	Item	Opens the Thread Information window showing system, test and user messages related to the current script	-	-	-
6	Go to line	Item	Opens the window for entering the script line and character number to go to	-	-	-

2.2.4.5 *The Message list menu elements*

The Message list menu contains commands for changing the message display parameters in the debugger window.

Table 2.2-9 describes the elements of the Message list menu.

Table 2.2-9

№	Element name	Element type	Description	Units	Default value	Value range
1	Clear	Item	Clears the Description field in the debugger window	-	-	
2	Colors	Item	Opens the window allowing to specify the words that a line should contain to be highlighted in color	-	-	-
3	Search	Item	Searches for a word in the Description window	-	-	-
4	Set filter	Item	Opens the window allowing to set the filtering criteria for the messages in the debugger window.	-	-	-
5	Apply filter	Item	Applies the current filter	-	-	-

2.2.5 The Filter dialog window

The Filter window allows to set the filtering criteria for the messages displayed in the Description field of the debugger window.

To open the Filter window, do one of the following:

1. Click the Edit test message ( ) button in the Editor-Debugger toolbar.
2. In the Debug and edit menu, select Edit test message.

Figure 2.2-6 shows the Filter window.

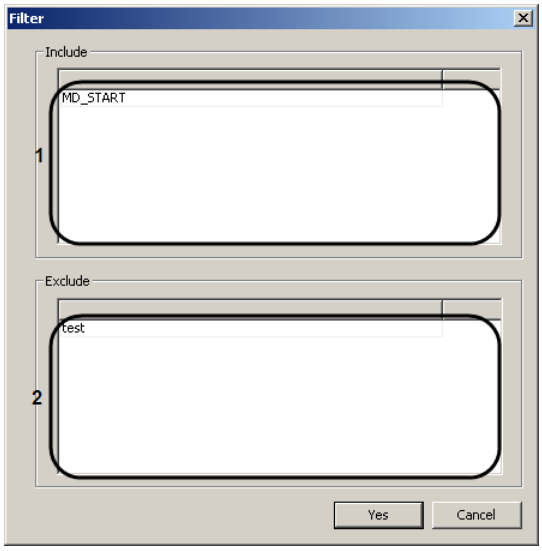

Figure 2.2-6. The Filter window

Table 2.2-10 describes the elements in the Filter window.

Table 2.2-10

№	Element name	Element type	Description	Units	Default value	Value range
1	Include	Text field	Only the lines containing the words from this field will be displayed in the debugger window	Latin, Cyrillic and special symbols	Empty	Unlimited number of characters
2	Exclude	Text field	The lines containing the words from this field will not be displayed in the debugger window	Latin, Cyrillic and special symbols	Empty	Unlimited number of characters

2.2.6 The Color dialog window

The Color dialog window is used for setting up color highlighting of the lines containing certain words.

To open the Color window, do one of the following:

1. Click the Colors ( ) button in the Editor-Debugger toolbar.

2. In the Message list menu, select Colors.

Figure 2.2-7 shows the Color window.

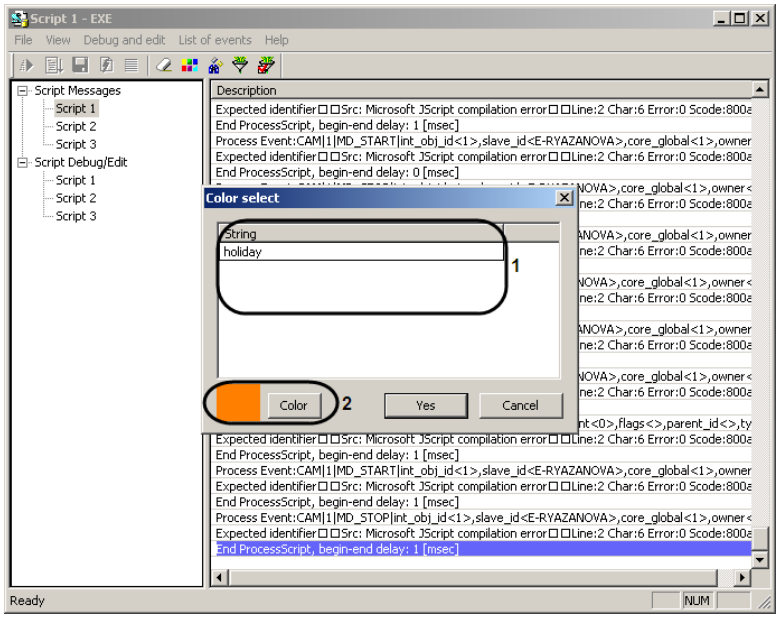

Figure 2.2-7. The Color window

Table 2.2-11 describes the elements in the Color window.

Table 2.2-11

№	Element name	Element type	Description	Units	Default value	Value range
1	Words	Text field	Contains the words or any sequences of characters, that make the line containing one of them, highlighted.	Latin, Cyrillic and special symbols	Empty	Unlimited number of characters
2	Color	Drop-down list	The color for highlighting the lines	RGB format	Grey	RGB colors

2.2.7 The toolbar of the Editor-Debugger utility

The Editor-Debugger toolbar is used for calling frequently used functions of the utility.

The toolbar operates in two modes: when the script control buttons are active, or when the debugger window control buttons are active.

The mode depends on the currently active tab of the Editor-Debugger utility: either the Script Debug/Edit tab for editing scripts, or the Script Messages tab for viewing messages in the debugger window.

Figure 2.2-8 shows the toolbar in the script editing mode.

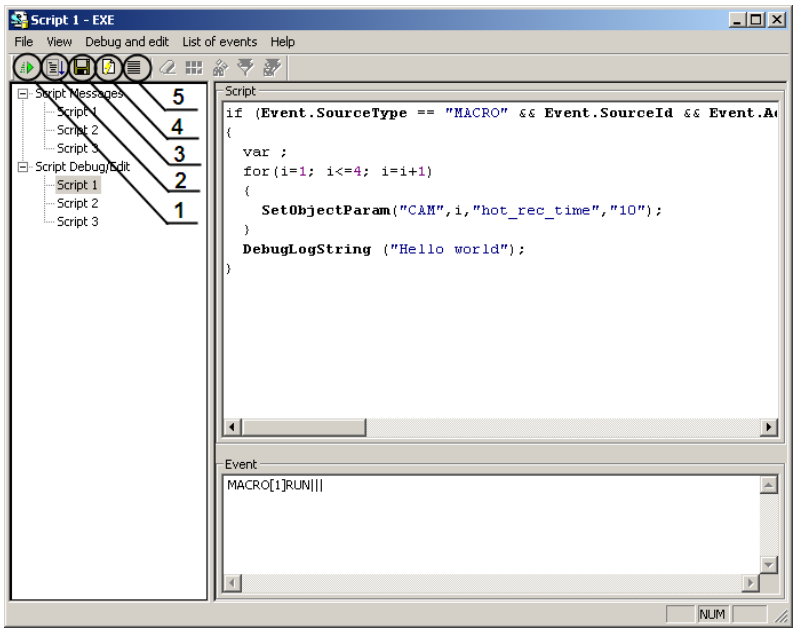

Figure 2.2-8. The toolbar in the script editing mode

Table 2.2-12 shows the elements of the toolbar in the script editing mode.

Table 2.2-12

№	Element name	Element type	Description	Units	Default value	Value range
1	Test run	Button	Runs the script with the test event	-	-	-
2	Test run in third-party debugger	Button	Runs the script using the third-party debugger	-	-	-
3	Save	Button	Saves the script to the Script object	-	-	-
4	Edit test event	Button	Opens the window for editing test events	-	-	-
5	Summary information	Button	Opens the Thread Information window that displays system, test and user messages related to the current script	-	-	-

Figure 2.2-9 shows the toolbar in the script debugging mode.

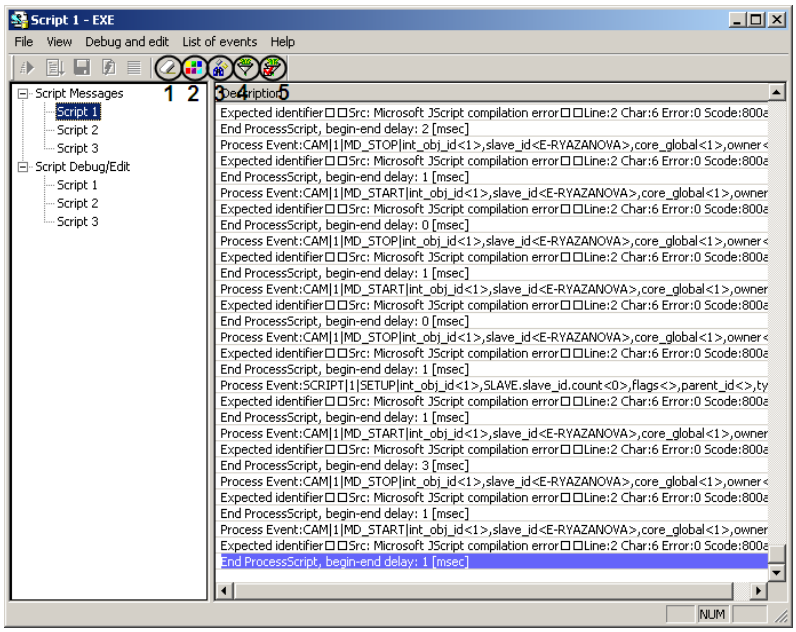

Figure 2.2-9. The toolbar in the debugger mode

Table 2.2-13 shows elements of the toolbar in the script debugging mode.

Table 2.2-13

№	Element name	Element type	Description	Units	Default value	Value range
1	Clear	Button	Clears the Description field in the debugger window	-	-	
2	Colors	Button	Opens the window allowing to specify the words that a line should contain to be highlighted in color	-	-	-
3	Search	Button	Searches for a word in the Description window	-	-	-
4	Set filter	Button	Opens the window allowing to set the filtering criteria for the messages in the debugger window.	-	-	-
5	Apply filter	Button	Applies the current filter	-	-	-