Programming Guide (JScript)

Programming in JScript

JScript functionality in Intellect

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

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

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

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

Description of the JScript object model in Intellect

The Core object and its built-in methods

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 reactions for them and changing their states. **Core** methods can pause script execution, script debugging, creating and calling global variables.

The **Core** object is not a prototype, thus no other objects can be created on its base (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.

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:

1. **objtype** - required argument. The type of the object whose parameters are to be set. It takes the following values: Type – String, range – existing object types.
2. **id** - required argument. Identification number of the object of the type set in the objtype parameter. It takes the following values: Type – String, range – existing object identification numbers of the specified type.
3. **param** - required argument. The parameter of the object. It takes the following values: Type – String, range – available parameters of the object.
4. **value** - required argument. The value to be set for the parameter specified in the param argument. It takes the following values: Type – String, range – depends on the parameter.

Usage examples

Example. When Macro 1 starts, check if Cameras 1 to 4 are set to broadcast color video. If a camera is set for black-and-white video broadcast, then switch it to the color mode (setting the true (“1”) value to the “Color” parameter – (“color”)).

```
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");
    }
  }
}
```

Note

If the object is active when the script is started (i.e. the setting panel of this object is open), then object parameters can not be changed by the SetObjectParam method. For instance, if the setting panel for the Camera 1 object is open and the aforecited script is started, the operation mode of Camera 1 will not be changed for the color one.

The SetObjectState method

The SetObjectState method changes the state of objects.

Method call syntax

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

Method arguments:

1. **objtype** - Required argument. The type of the object whose state is to be changed. It takes the following values: Type – String, range – existing object types.
2. **id** - Required argument. Identification number of the object of the type set in the objtype parameter. It takes the following values: Type – String, range – existing object identification numbers of the specified type.
3. **state** - Required argument. The state to switch the object to. It takes the following values: Type – String, range – available states of the object.

Usage examples

Example. 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");
  }
}
```

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. It 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);
}
```

The Base64Decode method

The Base64Decode method is used for decoding the lines that are coded by Base64 scheme.

Method call syntax

```
function Base64Decode(data_in: String, WideChar: Boolean)
```

Method arguments:

1. **data_in** - required argument. Set a line that should be decoded in Base64;
2. **WideChar** - required argument. Determines coding type. Can take 0 or 1 values. If coding type is Unicode, argument value is 1, otherwise 0.

Usage examples

Decode the line that is set in Base64 by starting macro №1. Output the decoding result into the debug windows of the Editor-Debugger utility. (Result is « Intellect JAVA SCRIPT» line).

```
if (Event.SourceType == "MACRO" && Event.SourceId == "1" && Event.Action == "RUN")
{
                var str = Base64Decode("SW50ZWxsZWN0IEpTY3JpcHQ= ", 0);
                DebugLogString(str);
}
```

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. It takes the following values: Type – int.

Usage examples

Problem 1. 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);
  }
}
```

Problem 2. When macro №2 starts, timer №1, that triggers every 10 seconds in a minute after macro №2 starting, starts.

> **Note**
>
> To start this script create the **Timer** object with ID=1 beforehand. Leave object parameters set by default (**Any**). The **Timer 1** object can be disabled

```
if (Event.SourceType == "MACRO" && Event.SourceId == "2" && Event.Action == "RUN")
{
for(i=0; i<=5; i=i+1)
{
DoReactStr("TIMER","1", "DISABLE", "");
Sleep(10000);
DoReactStr("TIMER","1", "ENABLE", "");
NotifyEventStr("TIMER","1", "TRIGGER", "");
}
DoReactStr("TIMER","1", "DISABLE", "");
}
```

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. It takes the following values: Type – String, satisfying the rules for the names of the string parameters in the Windows registry.

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 and 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", "10", "bright");
  }
  if (Event.SourceId == "2")
  {
    var cam10bright = Itv_var("cam10bright");
    for(i=1; i<=4; i=i+1)
    {
      SetObjectParam("CAM", i, "bright", cam10bright);
    }
  }
}
```

The Int_var method

The Int_var method sets and returns values of global variables of integer type.

Attention!

The Int_var method uses the same storage as the Itv_var method, but modify the type of variable to the integer type.

Method call syntax:

```
function  Int_var (globalvar : String) : int
```

Method arguments:

1. **globalvar** – required argument. The name of the global variable. It takes the following values: type – String, satisfying the rules for the names of the string parameters in the Windows registry.

Note.

Global variables are stored in the system 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 and HKEY_CURRENT_USER\Software\ITVScript\ ITVSCRIPT. To access a global variable directly from the registry, search the registry for it by the same name.

Usage examples. To check the method operation in the following test example the value 1 sets to the "2" global variable and then increases per 1 and displays on the script debug window.

```
if(Event.Action =="RUN")
{
    Int_var(2) = 1;
    Int_var("2")++;
    DebugLogString(Int_var("2").toString());
}
```

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. It takes the following values: Type – String, range – existing object types.

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);
  }
}
```

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:

1. **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. It takes the following values: Type – String, meeting the requirements for network computer names; range – computer names existing in the system.
2. **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. It takes the following values: Type – String; meeting the requirements for network computer names.

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);
}
```

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:

1. **objtype** - Required argument. The type of the object whose name is to be returned. It takes the following values: Type – String, range – existing object types.
2. **id** - Required argument. Identification number of the object of the type set in the objtype parameter. It 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>\Program 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.");
}
```

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:

1. **objtype** - Required argument. The type of the object whose state is to be returned. It takes the following values: Type – String, range – existing object types.
2. **id** - Required argument. Identification number of the object of the type set in the objtype argument. It 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");
  }
}
```

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:

1. **objtype** - Required argument. The type of the object whose parameter's value is to be returned. It takes the following values: Type – String, range – existing object types.
2. **id** - Required argument. Identification number of the object of the type set in the objtype argument. It takes the following values: Type – String, range – existing identification numbers of the objects of the specified type.
3. **param** - Required argument. The name of the parameter whose value is to be returned. It takes the following values: Type – String, range – available parameters of the object.

Usage examples

See the example for the SetObjectParam method.

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:

1. **objtype** - Required argument. The type of the object whose parent's identification number should be returned. It takes the following values: Type – String, range – existing object types.
2. **id** - Required argument. Identification number of the object of the type set in the objtype argument. It takes the following values: Type – String, range – existing identification numbers of the objects of the specified type.
3. **parent** - Required argument. The type of the object which is the parent of the object type specified by the objtype argument. It 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.SourceType == "CAM" && Event.Action == "DETACH")
{
    var cam_id = Event.SourceId;
    var parent_comp_id = GetObjectParentId("CAM", cam_id, "COMPUTER");
    DoReactStr("MAIL_MESSAGE","1","SETUP","from<***@mail.ru>,to<***@mail.ru>,body<Camera disabling "+cam_id+" on
the Server"+parent_comp_id+">,parent_id<1>,subject<Attention! Camera disabling>,name<Message 1>,objname<Message
1>");
    DoReactStr("MAIL_MESSAGE", "1", "SEND", "");
}
```

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:

1. **objtype** - Required argument. The type of the object that the action should be generated for. It takes the following values: Type – String, range – existing object types.
2. **id** - Required argument. Identification number of the object of the type set in the objtype argument. It takes the following values: Type – String, range – existing identification numbers of the objects of the specified type.
3. **action** - Required argument. The action to be generated. It takes the following values: Type – String, range – available actions for the objects of the specified type.
4. **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","REC","");
```

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.

For all reactions it is possible to specify delay of reaction performing using delay<> parameter. Delay is specified in seconds.

Note

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>");
}
```

Problem. When alarm by some camera is completed the record is to be continued for 5 second and after this time the record will be stopped (analogue of Post-record mode).

```
if (Event.SourceType == "CAM" &&  Event.Action == "MD_STOP")
{
    var camid = Event.SourceId;
    DoReactStr("CAM",camid,"REC_STOP","delay<5>");
}
```

Problem. Use macros 1 to enable telemetry control using mouse on the camera 4 displayed in the monitor 10. Use macros 2 to disable it.

```
if (Event.SourceType == "MACRO" &&  Event.Action == "RUN" &&  Event.SourceId == "1")
{
    DoReactStr("MONITOR","10","CONTROL_TELEMETRY","cam<4>,on<1>");
}
if (Event.SourceType == "MACRO" &&  Event.Action == "RUN" &&  Event.SourceId == "2")
{
    DoReactStr("MONITOR","10","CONTROL_TELEMETRY","cam<4>,on<0>");
}
```

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. It takes the following values: MsgObject objects created earlier in the script.

> **Note**
>
> 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);
  }
}
```

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

1. **objtype** - Required argument. The type of the object whose parameters are to be changed. It takes the following values: Type – String, range – existing object types.
2. **id** - Required argument. Identification number of the object of the type set in the objtype argument. It takes the following values: Type – String, range – existing identification numbers of the object of the specified type.
3. **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.

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) + ">");
  }
}
```

The DoReactSetup method

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

Method call syntax

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

Method arguments:

1. **objtype** - required argument. The type of the object the parameters of which are to be changed. It takes the following values: type – String, range – existing object types.
2. **id** - required argument. Identification number of the object of the type set in the objtype argument. It takes the following values: type – String, range – existing identification numbers of the objects of the specified type.
3. **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: values of the String type, range is limited by available parameters of the specified action. The “value” argument can take the following values: values of the String type, the range depends on the parameter.

Example. When Macro 1 starts, temporary remove all cameras on the first monitor.

```
 if (Event.SourceType == "MACRO" && Event.SourceId == "1" && Event.Action == "RUN")
{
DoReactSetup ("MONITOR","1","REMOVE_ALL","");
}
```

The DoReactGlobal method

The DoReactGlobal method is used to generate object reactions. The DoReactGlobal method transmits a reaction to the required object. The reaction is transmitted not only to the kernel, where the object is registered, but to the whole system. For the DoReactGlobal method the reaction is specified by the MsgObject object.

Method call syntax

```
 function  DoReactGlobal(msgevent : MsgObject)
```

Method arguments:

msgevent - required argument. It sets a reaction transmitted to the required object. It takes the following values: MsgObject objects created earlier in the script.

Example. When macro 2 starts, guard on Sensor 2. The command is to be transmitted to all system kernels as the reaction to be registered in the Events Log.

```
 if (Event.SourceType == "MACRO"&& Event.SourceId == "2" && Event.Action == "RUN")
{
var msgevent = CreateMsg();
```

```
msgevent.SourceType = "GRAY";
msgevent.SourceId = "2";
msgevent.Action = "ARM";
DoReactGlobal(msgevent);
}
```

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:

1. **objtype** - Required argument. The type of the object that the event should be generated for. It It takes the following values: Type – String, range – existing object types.
2. **id** - Required argument. Identification number of the object of the type set in the objtype argument. It takes the following values: Type – String, range – existing identification numbers of the object of the specified type.
3. **event** - Required argument. The event to be generated. It takes the following values: Type – String, range – available events for the objects of the specified type.
4. **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.

> **Note**
>
> 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", "");
}
```

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. It takes the following values: MsgObject objects created earlier in the script.

> **Note**
>
> Two types of system messages are available in Intellect system: events and reactions.
>
> The events usually contain some information and are used as notifications sent to all Intellect kernels connected to each other during the system configuration.
>
> The reactions are the control commands sent to specific objects.
>
> The reactions 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 reactions. The NotifyEventStr and NotifyEvent methods are used to generate events.

Example. When the Backup Archive 1 module starts archiving video recordings, the analog output 1 of the Video Capture Device 2 is disabled. Send the command as an event to be registered in the Events Log.

> **Note**
>
> While executing this script, the analog output 1 of the Video Capture Device 2 is not disabled

```
 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);
}
```

The NotifyEventGlobal method

The NotifyEventGlobal method is used to generate system events. The generated event is transmitted to all system kernels connected via the net. For the NotifyEventGlobal method the event is specified using the MsgObject object.

Method call syntax

```
function  NotifyEventGlobal (msgevent : MsgObject)
```

Method arguments:

msgevent - required argument. It sets the event transmitted to the system. It takes the following values: MsgObject objects created earlier in the script.

Example. When Macro 1 starts, the first camera is set for recording. The command is to be transmitted to all system kernels as the event to be registered in the Events Log.

> **Note**
>
> While executing this script, camera1 is not set for recording.

```
 if (Event.SourceType == "MACRO"&& Event.SourceId == "1" && Event.Action == "RUN")
{
var msgevent = CreateMsg();
msgevent.SourceType = "CAM";
msgevent.SourceId = "1";
msgevent.Action = "REC";
NotifyEventGlobal(msgevent);
}
```

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 1. 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);
}
```

Problem 2. When timer №1 starts, start macro №1 every 30 seconds.

Note

To start this script create the **Timer** object with ID=1 beforehand. Set value=1 to the **Second** parameter of the **Timer** object, leave other parameters without changing («**Any** by default)

```
if (Event.SourceType == "TIMER" && Event.SourceId == "1" && Event.Action == "TRIGGER")
{
var msg = CreateMsg();
 msg.StringToMsg(GetObjectParams("TIMER", "1"));
 if(msg.GetParam("s") == "1")
{
 DoReactStr("MACRO", "1", "RUN", "");
 SetObjectParam("TIMER","1","s","30");
DoReactStr("TIMER","1", "DISABLE", "");
DoReactStr("TIMER","1", "ENABLE", "");
 }
if(msg.GetParam("s") == "30")
{
```

```
 DoReactStr("MACRO", "1", "RUN", "");
               SetObjectParam("TIMER","1","s","1");
DoReactStr("TIMER","1", "DISABLE", "");
DoReactStr("TIMER","1", "ENABLE", "");
 }
}
```

The Lock and Unlock methods

The Lock and Unlock methods are used to create a global critical section when synchronization of scripts started in different streams is required. The Lock method opens a critical section and the Unlock method closes it.

> **Attention!**
>
> Attention! If the Lock method has been called, then the Unlock method has to be called too. Otherwise the system can freeze.

It is recommended to avoid using the Lock and Unlock methods.

Method call syntax

```
function Lock()
```

```
function Unlock()
```

Example. When Macro 1 starts, calculate total alarmed relays and sensors. Objects of each type are to be calculated at the same time (in an individual script). The result is to be stored to "counter" global variable.

Script 1:

```
 // Number of alarmed relays is calculated
var i = Number(0);
if (Event.SourceType == "MACRO" && Event.SourceId== "1" && Event.Action == "RUN")
{
```

```
var msg = CreateMsg();
msg.StringToMsg(GetObjectIds("GRELE"));
var objCount = msg.GetParam("id.count");
var k;
for(k= 0; k < objCount; k++)
if(GetObjectState("GRELE", msg.GetParam("id." + k))== "ALARM"){
                Lock();
                 i = Itv_var("counter");
i++;
Itv_var("counter")=i;
Unlock();
}
}
```

Script 2:

```
 //Number of alarmed sensors is calculated
var i = Number(0);
 if (Event.SourceType == "MACRO" && Event.SourceId== "1" && Event.Action == "RUN")
{
var msg = CreateMsg();
msg.StringToMsg(GetObjectIds("GRAY"));
var objCount = msg.GetParam("id.count");
var k;
for(k = 0; k < objCount; k++)
if(GetObjectState("GRAY", msg.GetParam("id." + k))== "ALARMED"){
                Lock();
                 i = Itv_var("counter");
i++;
Itv_var("counter")=i;
Unlock();
}
}
```

Note

If the Lock() and Unlock() methods are not used in this example, then collisions may occur and calculated value will be less than a real one

The IsAvailableObject method

The IsAvailableObject method is used to determine the current access rights to an object.

Method call syntax:

```
function  IsAvailableObject(compname: String, objtype: String,  id: String,  param : String) : String
```

The method returns 0 if the current user has not been assigned access rights of type **param** for the object; it returns 1 if these rights have been assigned.

Method arguments:

1. **compname** - required. Corresponds to the name of the **Computer** object on which the object was created in the hardware tree.
2. **objtype** - required. Corresponds to the type of the system object to which access rights are are being checked. Allowed values: String type; the set of values is restricted to the object types registered in the system.
3. **id** - required. Corresponds to the identification (registration) number of the object of the type specified by the **objtype** argument. Allowed values: String type; the set of values is restricted to the identification numbers of objects of the specified type, which are registered in the system.
4. **param** - required. Corresponds to the number of the type of rights which are being checked. A description of rights is given in Limiting access to the administration of the system objects, control and viewing functions of the Administrator's Guide. Allowed values:
    1. 0 - NoView access rights. The method will return 1 if there are no administrative, control-, or viewing rights for the object (red 'x').
    2. 1 - NoControl access rights. The method will return 1 if there are only viewing rights (letter M).
    3. 2 - ViewAndControl access rights. The method will return 1 if there are control- and viewing rights for the object (green box).
    4. 3 - ViewOrControl access rights. The method will return 1 if there is either viewing or control rights for the object.
    5. 4 - Not access rights (e.g. no access rights).
    6. 5 - Configure access rights. The method will return 1 if there are administrative, control- and viewing rights for the object (gray box).

For example: Based on a **Computer** object named "S-UYUTOVA", a **Camera** object with the identifier 1 has been created in the hardware tree. To determine the current access rights to the object:

```
var i = 0;

for(i = 0; i <= 5; i++)

{

     var result =
```

```
    IsAvailableObject('S-UYUTOVA','CAM','1', i);

    DebugLogString("right "+i+" = "+result);

}
```

The GetUserId method

The GetUserId method returns the identifier of the current *Intellect* Software System user.

Method call syntax:

```
function  GetUserId (cmp : String) : String
```

Method arguments:

1. **cmp** - required. Specifies the name of the computer on which the *Intellect* Software System is installed. Allowed values: values of String type that satisfy the requirements for computers' network names; the set of values is restricted to the names of computers registered in the system.

For example: To display the identifier of the current user of the *Intellect* Software System, which has been installed on a computer named 'WS3':

```
DebugLogString(GetUserId("WS3"));
```

The GetEventDescription method

The GetEventDescription method is in use for receiving the object description on free language.

Syntax of method invocation:

```
function  GetEventDescription (obj_type : String, event : String)
```

Method arguments:

1. **obj_type** – required argument. Specifies type of system object description of which is required to get.
2. **event** – required argument. Specifies the name of object description of which is required to get.

Example. Display messages about receiving of events for camera 1 on the free language in the Debug window.

```
if (Event.SourceType == "CAM"&& Event.SourceId == "1")
{
        var str = GetEventDescription("CAM", Event.Action);
        DebugLogString(str);
}
```

The GetObjectIdByParam method

The GetObjectIdByParam method allows receiving of object ID at which some parameter is equal to the specified value. The first found object ID is receiving if there are several such objects. The null is receiving if such objects are not found

Syntax of method invocation:

```
function GetObjectIdByParam (obj_type : String, obj_param : String, param_value : String)
```

Method arguments:

1. **obj_type** – required argument. Specifies type of system object, ID of which is required to get.
2. **obj_param** - required argument. Specifies the name of parameter in database on value of which the object is to be found.
3. **param_value** – required argument. Specifies the required value of object parameter.

Example. Find cameras from which black-and-white image is receiving and set the **Color** parameter equals to 1 for them.

```
if (Event.SourceType == "MACRO" && Event.SourceId== "1" && Event.Action == "RUN")

{
  var id = GetObjectIdByParam("CAM","color","0"); //receiving the first object ID
  while (id){ //while exist the Camera objects from which black-and-white image is receiving
     SetObjectParam ("CAM", id, "color", "1"); //change the Color parameter for found object
     id = GetObjectIdByParam("CAM","color","0"); //receiving the next object ID
  }
}
```

The SaveToFile method

SaveToFile method is in use to save in file the frame from camera receiving in the data parameter of FRAME_SENT event.

Syntax of method invocation:

```
function  SaveToFile (path: String,  data: String,  param : Boolean)
```

Saving of frame is also can be performed using the GET_FRAME reaction of the CAM object. It is required to specify the path for saving file with frame in the path parameter of this reaction. FRAME_SENT event is created in the system if the GET_FRAME reaction has not got the path parameter. In the data parameter of the FRAME_SENT event the video image frame which is to be saved using the SaveToFile method is stored.

The reaction allows exporting of video image frame even if the camera is not displaying in the Monitor window.

Method argument:

1. **path** – mandatory argument. Specifies the full path to save the file with frame.
2. **data** – mandatory argument. Specifies data to save in file.
3. **param** – mandatory argument. Defines necessity of conversion from base 64 format before saving. Possible parameter values:
    1. true – decode from base64 before saving;
    2. false – save string without conversion.

Time of frame saving depends on the anchor frames frequency. The higher the anchor frames frequency, the less time.

Example. Save the frame receiving from Camera 1 in the test.jpg file on the D disk:

```
if (Event.SourceType == "CAM" && Event.SourceId == "1" && Event.Action == "FRAME_SENT")
  {
    SaveToFile("D:\\test.jpg",Event.GetParam("data"),true);
  }
```

The GetLinkedObjects method

The GetLinkedObjects method is used to get list of objects linked to the specified camera using the Objects connection object (see the Administrator's Guide, Connection of objects with cameras section)

Method call syntax:

```
function  GetLinkedObjects(type1 : string, id : string, type2 : string)
```

Method argument:

1. **type1** – the type of object for which linked objects are to be returned.
2. **id** – identification number of object for which linked objects are to be returned.
3. **type2** – the type of linked objects which are to be returned. If empty string is sent then linked objects of all types will be returned.

Example.

The **Objects connection** object is configured the following way:

Display in debug window the list of objects linked with the camera 1.

```
if (Event.SourceType == "MACRO")
{

varmsgstr = GetLinkedObjects("CAM","1","MACRO")

DebugLogString("Linked objects  " + msgstr);

}
```

As a result the "Linked objects MACRO:1" message will be displayed in the script debug window.

The WriteIni method

The WriteIni method is used for writing the string variable to ini-file.

Method call syntax:

```
function WriteIni(varName: String, varValue: String, path: String)
```

Method argument:

1. **varName** – required argument. Sets name of variable for saving in the file.
2. **varValue** – required argument. Sets value of t variable.

3. **path** – required argument. Sets full path to the ini-file in which variable is to be stored. Storage of variables can be replaced on the network resource. Enter the network path in the argument for it.

Example. Write the MyVar variable to the \\fs\temp\test.ini file and specify the «Helloworld!” value to it. Then read the written value and display it on the script debug window.

```
WriteIni("MyVar", "Hello world", "\\\\fs\\temp\\test.ini");

var result = ReadIni("MyVar","\\\\fs\\temp\\test.ini");

DebugLogString(result);
```

The ReadIni method

The ReadInimethod is used to read values of string variablein the ini-file.

Method call syntax:

```
function ReadIni (varName: String, path: String): String
```

Method call syntax:

1. **varName** – required argument. Sets name of the variable storing in the file.
2. **path** – required argument. Sets the full path to the ini-file in which variable is storing.

See example in The WriteIni method section.

The AddIni method

The AddIni method is used to write, change and read integer variable from the ini-file. The method returns the value of variable after its changing.

Method call syntax:

```
function AddIni(varName: String, varValue: int, path: String): int
```

1. varName – required argument. Sets the name of variable in the file.
2. varValue – required argument. Sets the value of variable or value which should be added to the existing value of variable:
    1. The varValue value will be assigned to the variable if there is a file with the varName name and string value in the file.
    2. If there is no variable with the varName name in the file then such variable will be created and the varValue value will be assigned to it.

3. If there is a variable with the varName name in the file and it has integer value or its value is indicated to the integer type, then value will be indicated and the varValue value will be added to it.

3. path – required argument. Sets the full path to the ini-file in which variable it to be stored. Storage of variables can be placed on the network resource. Enter the network path in the for it.

Example. There is no the "MyVar" variable in the "C:\\test.ini". Write such variable with the -1 value to the file, then add 1 to it and display the result value on the script debug window.

```
var result = AddIni("MyVar", -1, "C:\\test.ini");

result = AddIni("MyVar", 1, "C:\\test.ini");

DebugLogString(result);
```

The SetTimer method

The SetTimer method is used to start the timer.
The syntax for accessing the method is:

```
function SetTimer (id : int, milliseconds : int)
```

Method arguments:

1. **id** is a required argument. It specifies the timer ID. Allowed values are int or string type.
2. **milliseconds** is a required argument. It specifies the period with which the timer will trigger if it is not stopped by the KillTimer method. It is specified in milliseconds. Allowed values are int type.

Example. 2 seconds after the macro 1 is executed, start recording on camera 1.

```
if(Event.SourceType=="LOCAL_TIMER" && Event.Action=="TRIGGERED" && Event.SourceId==333) //it is possible to use
Event.SourceId == "333", i.e. string ID
{
  var actuallyKilled = KillTimer(333);
  if(actuallyKilled == 1)
  {
     DoReactStr("CAM","1","REC","");
  }
}
```

```
if(Event.SourceType=="MACRO"&& Event.SourceId == "1" && Event.Action == "RUN")
{
  SetTimer(333,2000); //333 - id, 2000 msec = 2 sec - period
}
```

The KillTimer method

The KillTimer method is used to stop the timer. Returns 1 if timer was stopped as a result of function executed.
The syntax for accessing the method is:

```
function KillTimer (id : int) : int
```

Method arguments:

1. **id** is a required argument. It specifies the timer ID. Allowed values are int or string type.

Example. See in the The SetTimer method section.

The GetObjectChildIds method

The GetObjectChildIds method returns IDs of child objects of specified type created under given object.

Method call syntax

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

Method arguments:

1. **parent** - Required argument. The type of the object for which you want to find out child objects . It takes the following values: Type – String, range – existing object types.
2. **id** - Required argument. Identification number of the object of the type set in the **parent**argument. It takes the following values: Type – String, range – existing identification numbers of the objects of the specified type.
3. **objtype** - Required argument. The type of the child objects whose identification numbers should be returned. It takes the following values: Type – String, range – existing object types.

Example. Arm all cameras on WS2 computer on Macro 1.

```
if (Event.SourceType == "MACRO" && Event.Action == "RUN" && Event.SourceId == "1")
{
   var children = GetObjectChildIds("SLAVE","DESKTOP-UBOS6BK","CAM");
```

```
    ch=children.split(",");
    for (i=0;i<ch.length; i++ )
    {
      DoReactStr("CAM",ch[i],"ARM","");
    }
}
```

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

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.

The GetSourceType method

The GetSourceType method returns the type of the **MsgObjec**t or **Event** object.

Method call syntax

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

Method arguments: no arguments.

Usage examples

Problem. When Macro 1 starts, arm Detection Zones *.1 in the Day mode for Cameras № 1 – 4. When Macro 2 starts, arm Detection Zones *.2 in the Night mode for Cameras № 1 – 4. When Macro 3 starts, arm 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 arming the *.1 detection zones
  if(Event.GetSourceId() == "1")
  {
for (k = 1;  k<= 4;  k = k+1)
    {
      DoReactStr("CAM_ZONE", k + ".1", "ARM", "");
      DoReactStr("CAM_ZONE", k + ".2", "DISARM", "");
      DoReactStr("CAM_ZONE", k + ".3", "DISARM", "");
    }
  }

  //Switching the cameras to the Nigh mode by arming the *.2 detection zones
  if(Event.GetSourceId() == "2")
  {
    for (k = 1;  k<= 4;  k = k+1)
    {
      DoReactStr("CAM_ZONE", k + ".1", "DISARM", "");
      DoReactStr("CAM_ZONE", k + ".2", "ARM", "");
      DoReactStr("CAM_ZONE", k + ".3", "DISARM", "");
    }
  }

  //Switching the cameras to the Rain mode by arming the *.3 detection zones
  if(Event.GetSourceId() == "3")
  {
    for (k = 1;  k<= 4;  k = k+1)
    {
      DoReactStr("CAM_ZONE", k + ".1", "DISARM", "");
      DoReactStr("CAM_ZONE", k + ".2", "DISARM", "");
      DoReactStr("CAM_ZONE", k + ".3", "ARM", "");
    }
  }
}
```

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.

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.

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). It takes the following values: type – String, range – available parameters for the objects of the specified type.

Note

If the object has no parameter with this name, then the method restores an empty string

Example. When any event from any camera is registered, check from which computer the event comes. If the computer has “WS3” name, create the event copy where the computer’s name is “Computer”.

```
 if (Event.SourceType == "CAM")
{
var msg = Event.Clone();
if (msg.GetParam("slave_id") == "WS3")
{
msg.SetParam("slave_id", "Computer");
NotifyEvent(msg);
}
}
```

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:

1. **param** - Required argument. The name of the parameter of the object created using **MsgObject** (or of the **Event** object). It takes the following values: Type – String, range – available parameters for the objects of the specified type.
2. **value** - Required argument. The value to be assigned to the parameter specified in the param argument. It takes the following values: Type – String, range – depends on the parameter.

Usage examples

See the example for the GetParam method.

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", "");
}
```

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. It 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);
}
```

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. It 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|"

Usage examples

Example. Upon registration the connection ("Attach") event for any camera, in the system retrigger the "Attach" event with modified **Number of the PTZ device** (telemetry_id) and **N umber of the microphone for synchronous recoeding** (audio_id) parameters. 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);
 }
}
```

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","");

  }
}
```

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", "");
}
}
```

The GetObjectParams method

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

Method call syntax:

```
 function GetObjectParams(objectType : String, objectId : String)
```

Method arguments:

1. **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.
2. **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", "");
}
}
```

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>");
}
```

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.

The Action property

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

Property call syntax

```
SourceId : String
```

Usage examples

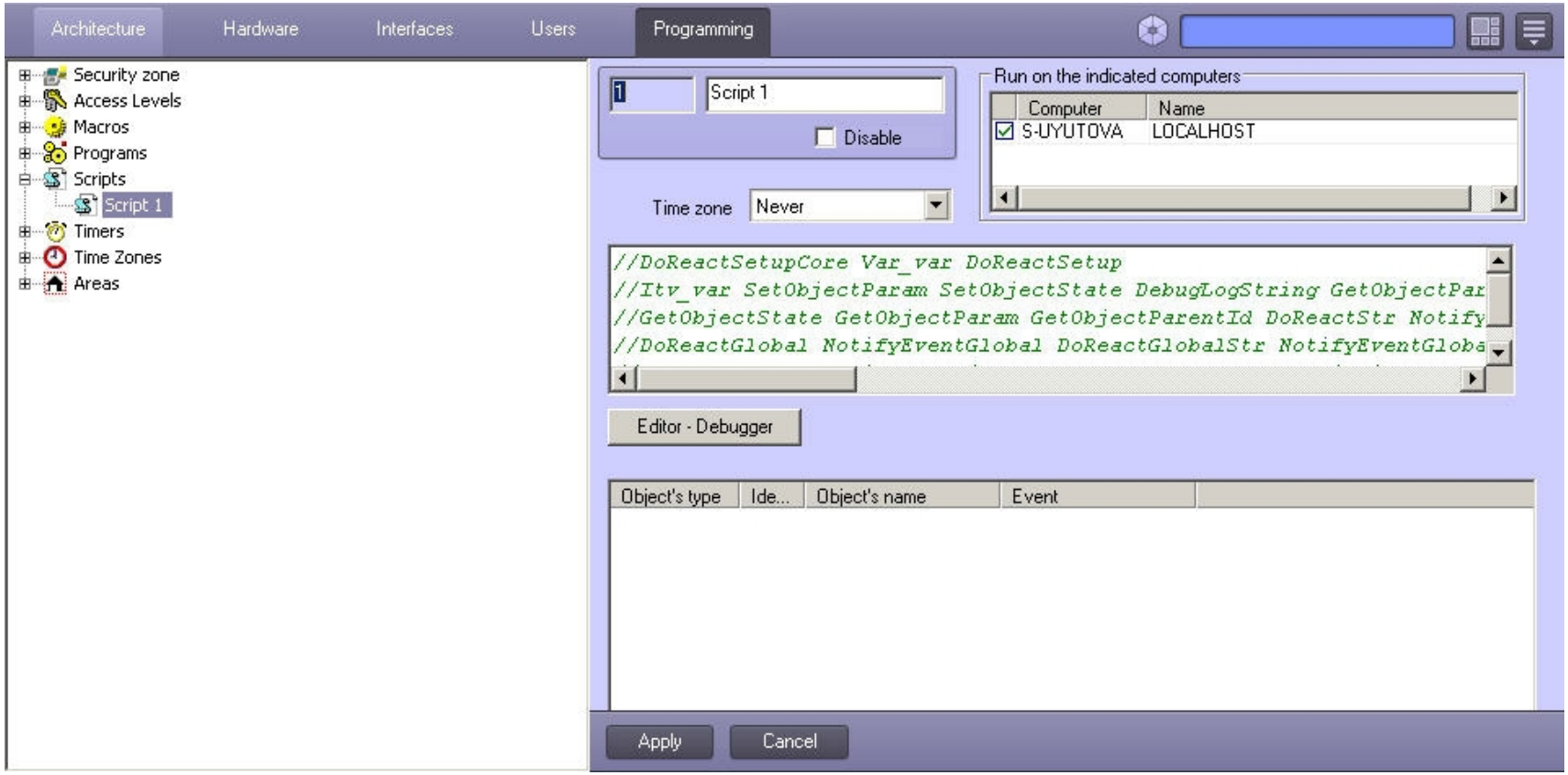
See the example for the SourceType property.

Programming tools

The Script object

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

Figure shows the settings panel for the **Script** object.



Attention!

Creating of large number of the **Script** objects (more than 100) can result to instable system working.

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

Note.

To set all checkboxes checked select one in the column and click Ctrl+A. To set all checkboxes unchecked select one and click Shift+A.

It also has the button for starting the *Editor-Debugger* tool and the text window for viewing the script text created by means of this tool. The scripts can be edited in the *Editor-Debugger* tool or directly in the settings panel for the **Script** object.

Moreover, one can configure the filter of events – the list of events which are to be processed by the **Script** system object. In general, including the event to the filter equals to *if* operator in the text of script, i.e. when the event is in the table, the operator can be omitted.

Attention!

The event filter is to be set when creating a script in large distributed configurations. Otherwise the module will process all incoming events and it will lead to module malfunctioning.

Example

If the **Object's type** column has the **Macro** value, the **Identifier** column has the **1** value and the **Event** column has the **Executed** value, then

```
if (Event.SourceType == "MACRO" && Event.SourceId==1 && Event.Action == "RUN")
{
    DoReactStr("CAM", "2", "REC", "");
}
```

script can be used instead of script.

```
DoReactStr("CAM", "2", "REC", "");
```

The detailed information on the elements of the settings panel for the **Script** object is given in Administrator's guide.

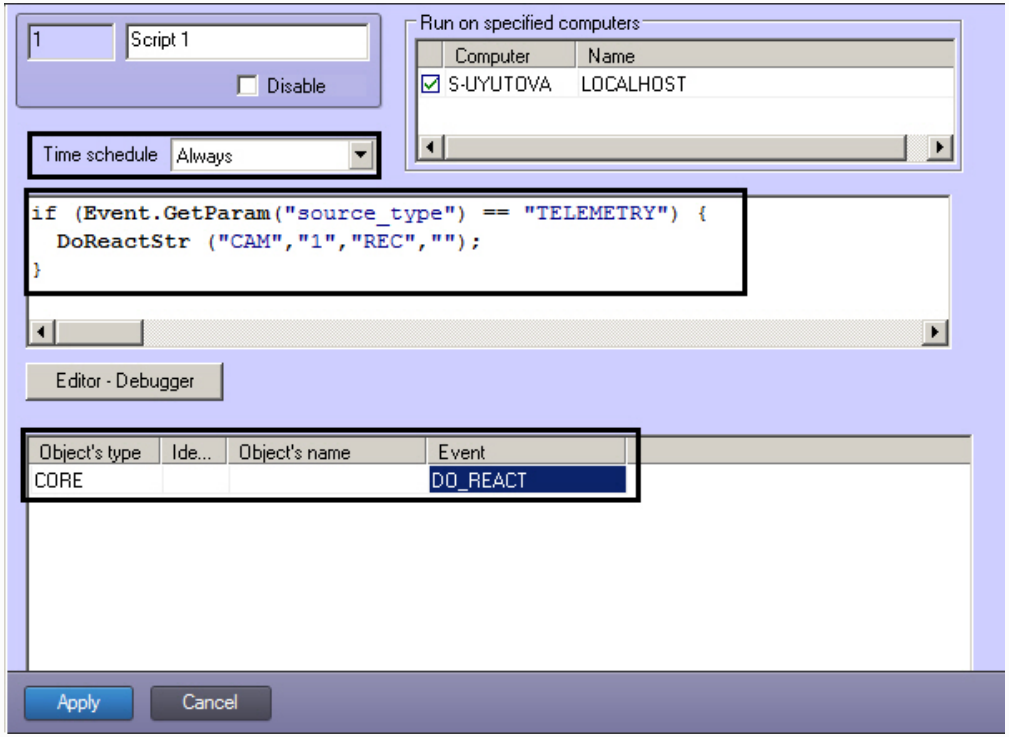
Example.

Recording on Camera 1 is to be started when controlling PTZ device in the Video surveillance monitor.

For this adjust the **Script** object as follows:

1. Select the required time zone when the script is to be executed.

2. Enter the script text:



3. Adjust the filter as follows:
    1. Select CORE in the **Object's type** dropdown list.
    2. Enter DO_REACT in the **Event** field.

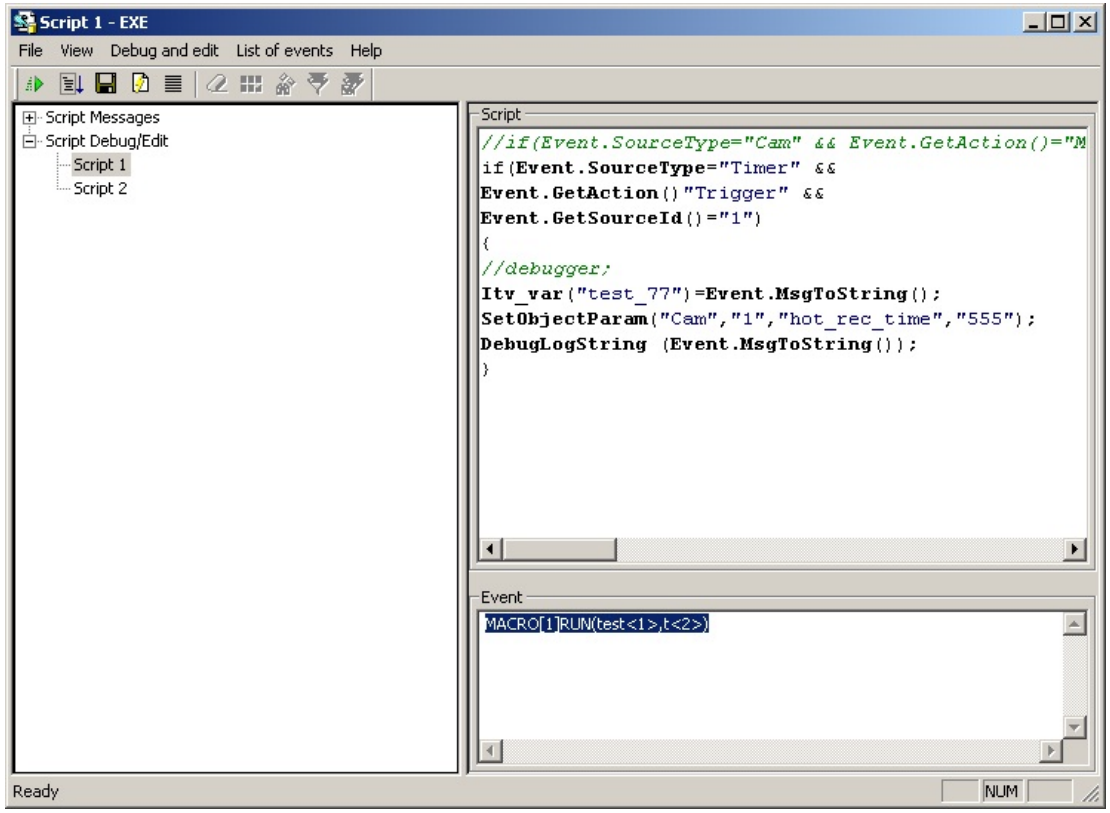
The filter can be set by the UPDATE_OBJECT event of the CORE object. Example command to add **Camera** 1 object to the **Script** 2 filter:

```
NotifyEventStr("CORE","","UPDATE_OBJECT","objtype<SCRIPT>,objid<2>,EVENT.objid.0<1>,EVENT.objid.1<10>,EVENT.action.count<2>,flags<>,EVENT.action.0<>,EVENT.
action.1<>,EVENT.objtype.0<CAM>,EVENT.objtype.count<2>,EVENT.objtype.1<CAM>,EVENT.objid.count<2>");
```

The Editor-Debugger utility

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

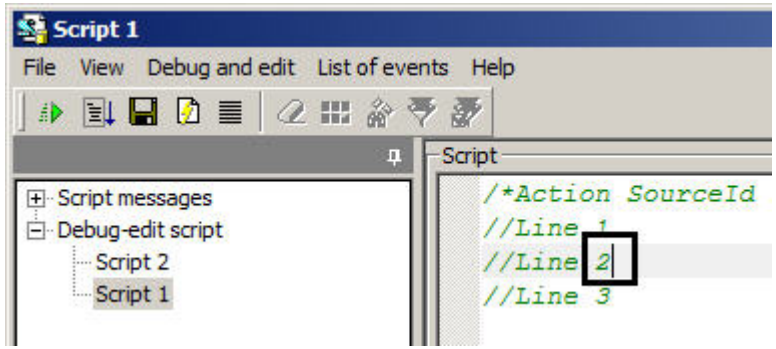
Figure shows the *Editor-Debugger* dialog window.



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.

> **Note.**
>
> The position of the cursor in each script is stored in the context of one Intellect session (the cursor position is reset at the product restart). For example, if you place the cursor at the end of **//Line 2** in **Script 1**, then switch to **Script 2** and perform any action in it, the cursor will be still at the end of **//Line 2** when you return to **Script 1**.



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.

The Debug window

Intellect allows viewing all events and reactions happening in the system in real time mode. Events and reactions with object properties are displayed in the **Debug window**. They can be copied to the Windows clipboard and then used in programs.

Enabling the Debug window

By default the debug window is disabled. To enable the debug window, do the following:

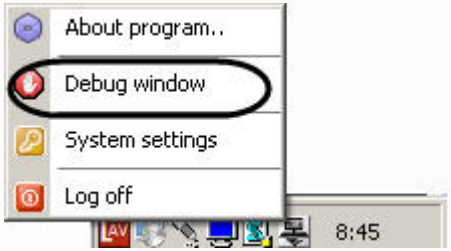
1. Shut down *Intellect.*
2. Run the *Tweaki.exe* utility.

> **Note.**
>
> The Debug window can be enabled without the tweaki.exe utility. For this set values 1, 2 or 3 for the Debug string parameter in the HKEY_LOCAL_MACHINE\SOFTWARE\ITV\Intellect section of the Windows registry (HKEY_LOCAL_MACHINE \Software\Wow6432Node\ITV\Intellect for 64-bit system).

3. Select **Intellect** section in the tree on the left side of the utility dialog box.
4. Change the value of the **Debug mode** parameter from **Disabled to Debug 1**, **Debug 2** or **Debug 3**.
5. Click the **OK** button.
6. Start *Intellect.*
7. A new **Debug window** item appears on the Main control panel of *Intellect*.

Note.

This menu is also available in the Windows notification area (system tray) – left click on the  button or short click the F8 hot key.



8. Select the **Debug window** item on the Main control panel in order to display the Debug window on the monitor. The selected **Debug window** item is marked with a flag.

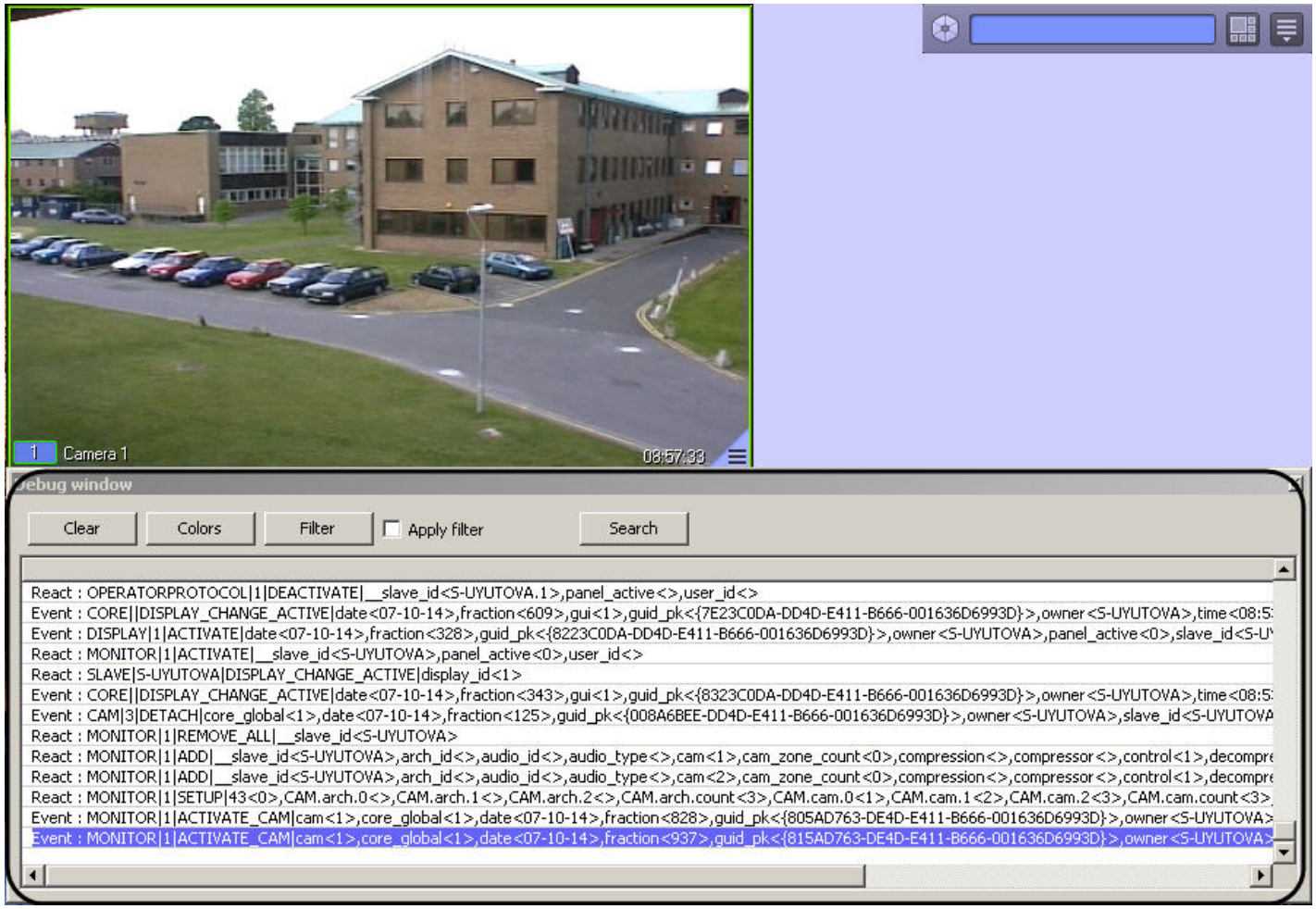
To hide the Debug window, re-select the Debug window item on the Main control panel.

Note.

To disable the Debug window, select the **Disable** value for the **Debug mode** in the tweaki.exe utility or set value 0 for the Debug string parameter in the HKEY_LOCAL_MACHINE\SOFTWARE\ITV\Intellect section of the Windows registry (HKEY_LOCAL_MACHINE \Software\Wow6432Node\ITV\Intellect for 64-bit system). These actions are performed when *Intellect* is exited.

Working with Debug window

Look at the figure of the Debug window. The Debug window displays the sequence of events and reactions in the system.



Here are the features of the Debug window:

1. always on top;
2. the size of the Debug window is changed using the mouse;
3. information on event or reaction can be copied to the Windows clipboard and then used in programs;
4. events and reactions can be filtered in the Debug window;

5. events and reactions can be highlighted in the Debug window;
6. it is possible to search for events or reactions in the Debug window.

Regular expressions can be used in order to highlight and filter messages in the Debug window.

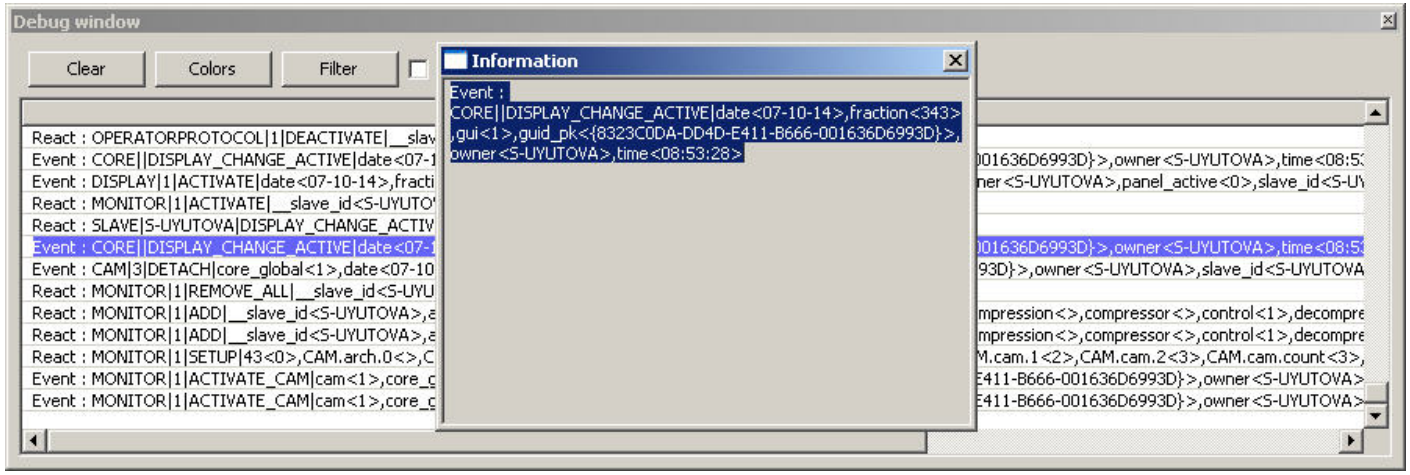
Copying information on event or reaction to the clipboard

To read and/or copy data on event or reaction to the Windows clipboard, do the following:

1. Highlight the required line in the **Debug window**.
2. Right-click on the highlighted line. The **Info** window with info on required event or reaction appears.
3. Highlight the information that is to be copied to the Windows clipboard and click **Ctrl**+**C**.

> **Note.**
>
> Use the context menu for operations with text in the **Info** window (right-click on the highlighted text).

4. To close the **Info** window, click the  button.

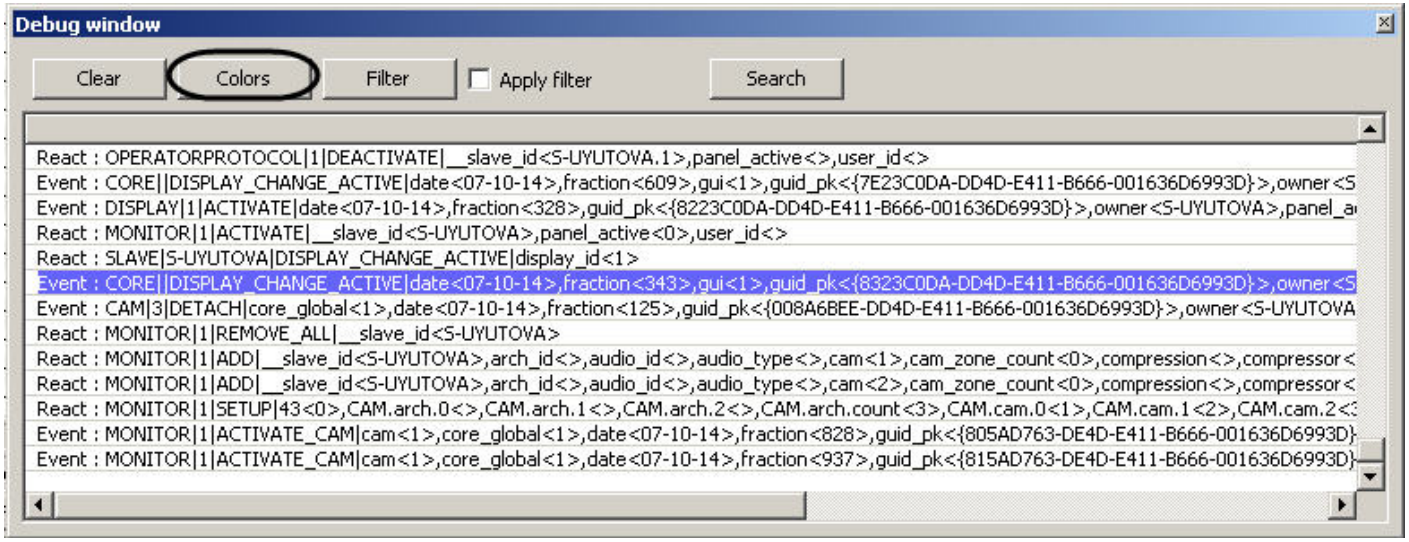


Information on event or reaction is now copied to the Windows clipboard.

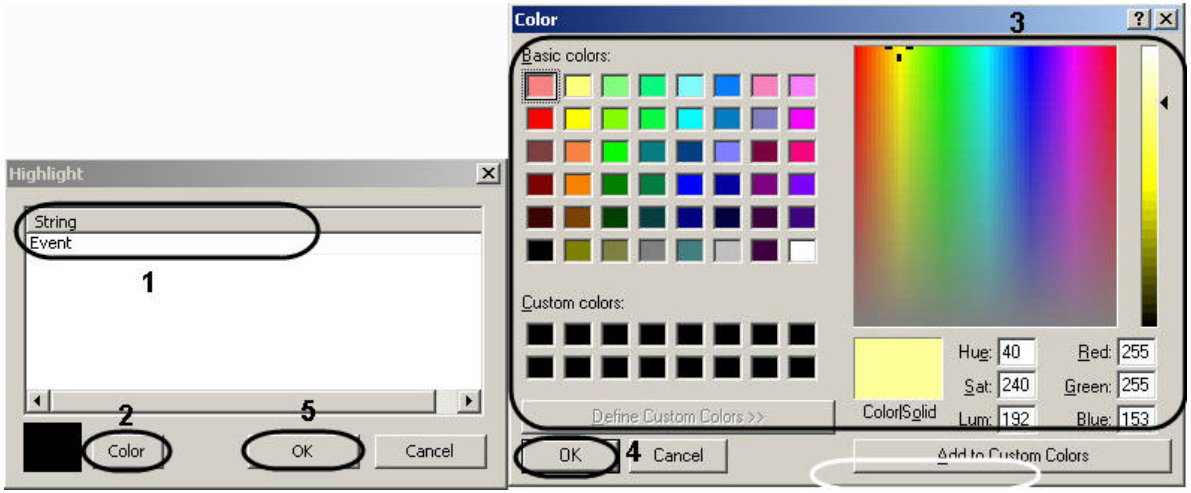
Highlighting messages

To configure message highlighting in the Debug window, do the following:

1. Click the **Colors** button.



2. In the **Highlight in** window specify the line that is to be highlighted every time it appears in the message (1).

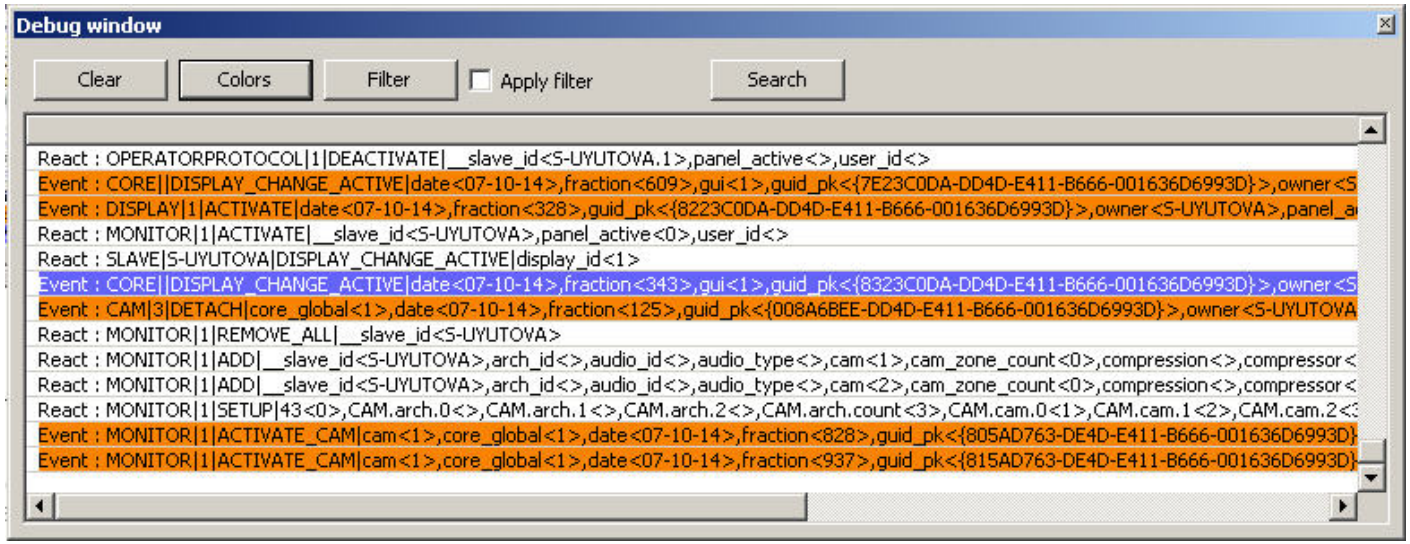


3. Click the **Color** button (2).
4. Select the color in the **Color** common dialog box (3).
5. Click the **OK** button(4).
6. Repeat steps 2-5 for all required lines.

> **Note.**
>
> To add a line to the table, click the key on the keyboard.

7. Click the **Yes** button (5).

As a result messages with the specified line are highlighted in the Debug window.



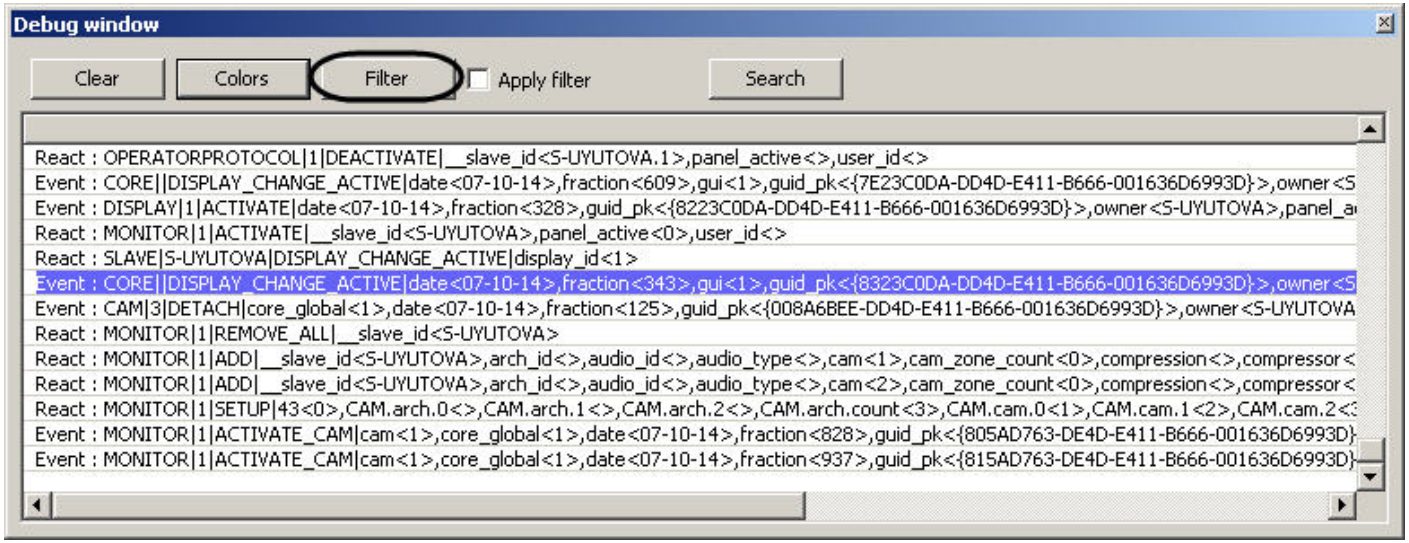
The messages are now highlighted.

Event and reaction filter

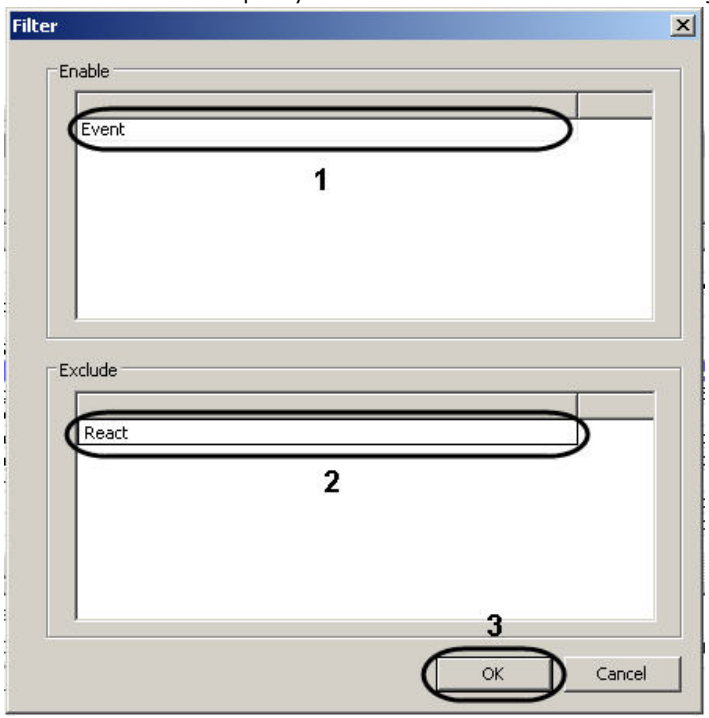
The filter of events and reactions allows displaying only required messages in the Debug window.

To configure the filter of events and reactions, do the following:

1. Click the **Filter** button**.**



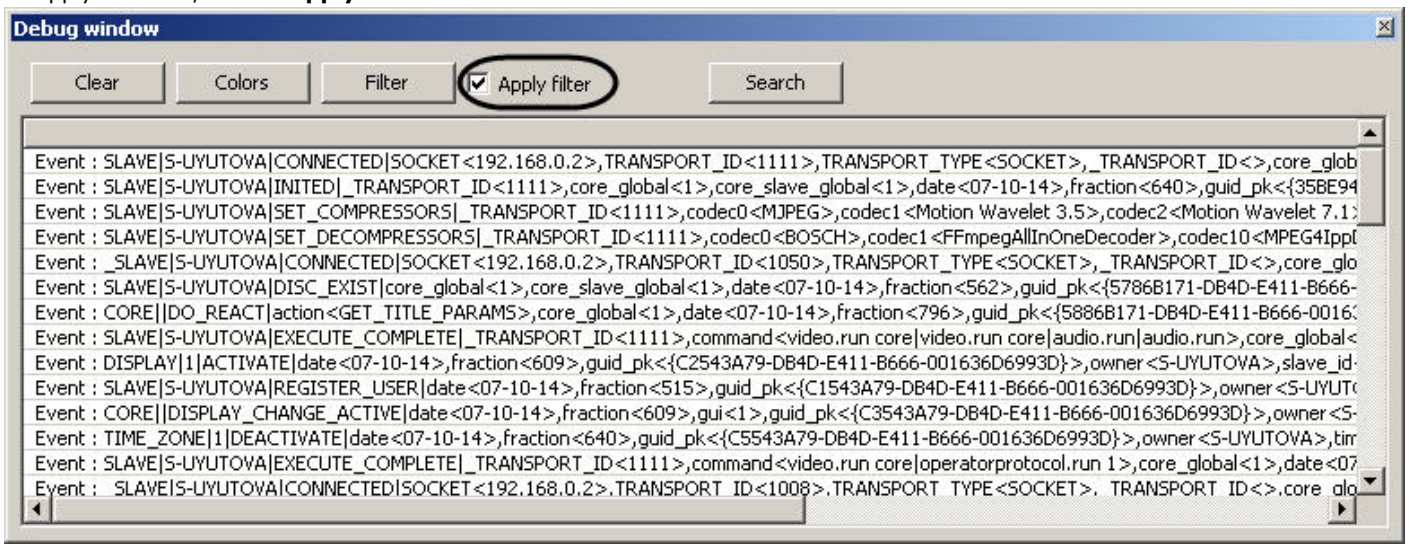
2. In the **Filter** window specify the lines that are to be in the message for it to be displayed in the Debug window (1).



3. Specify the lines that are to be in the message for it to not be displayed in the Debug window (2).

> **Note.**
>
> To add a line to the table, click the key on the keyboard.

4. Click the **Yes** button (3).
5. To apply the filter, set the **Apply filter** checkbox checked.



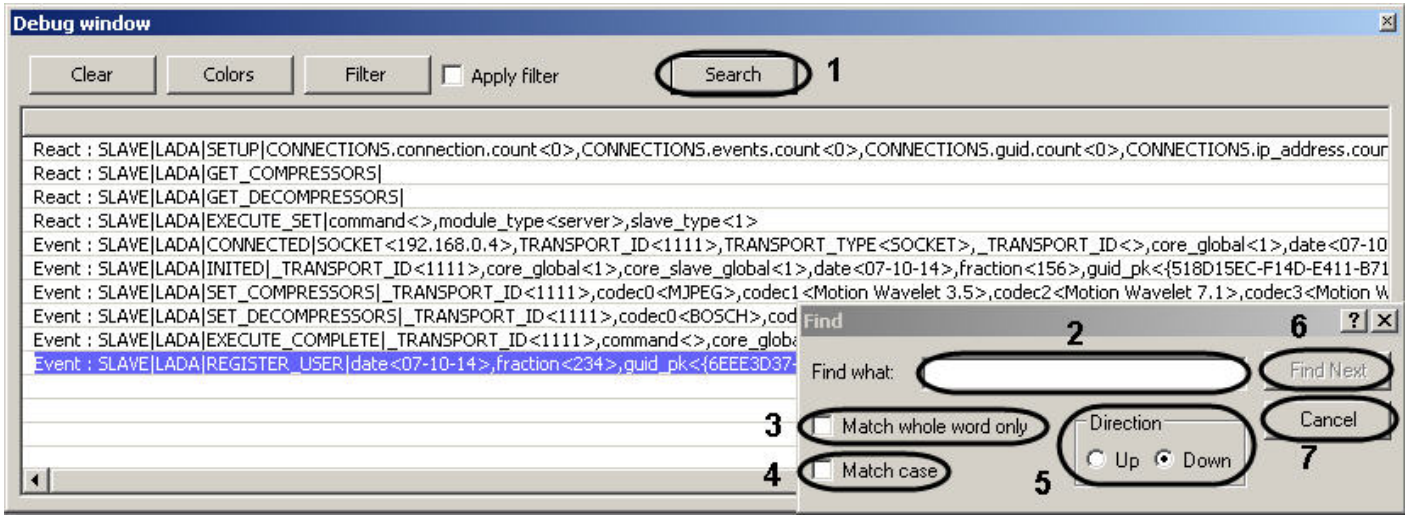
As a result only messages that fit the search criteria are displayed in the Debug window.

The event and reaction filter is now configured.

Searching for events and reactions

To search for events and reactions, do the following:

1. Click the **Search** button (1). The **Search** window appears.



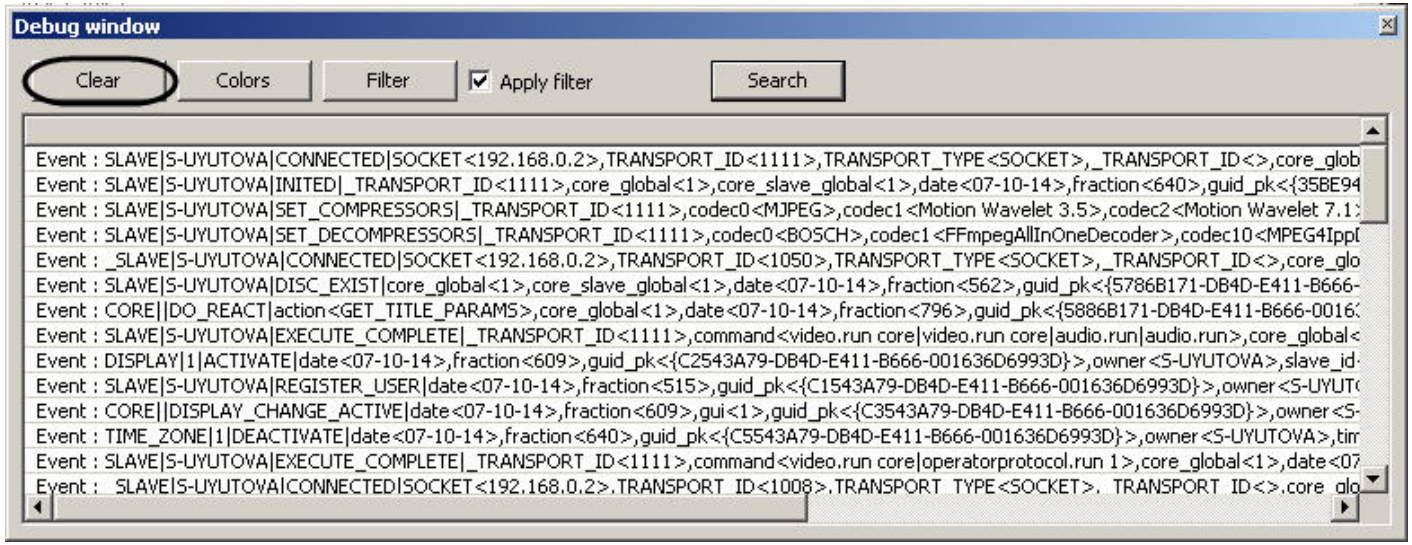
2. Specify the search criteria in the **Search for:** field (2).
3. If the specified string is to be searched as a whole word, not a part of any other word, then set the **Match whole word** checkbox checked (3).
4. If the search is to be case sensitive, then set the **Case sensitive** checkbox checked (4).
5. Set the **Direction** switch into the position corresponding to the search direction (5).
6. To view the following search result, click the **Find next** button (6).

> **Note.**
>
> To close the **Search** window click the **Cancel** button.

The search for events and reactions is now complete.

Clearing the Debug window

To delete all messages from the Debug window, click the **Clear** button.



Getting the list of object names, reactions and events in INTELLECT™

The list of object names, reactions and events used while programming in *INTELLECT*™ software can be got with the help of ddi.exe tool.

ddi.exe tool is started in one of the following ways:

1. "**Start All Programs Intellect Tools System configuration**.

   Note

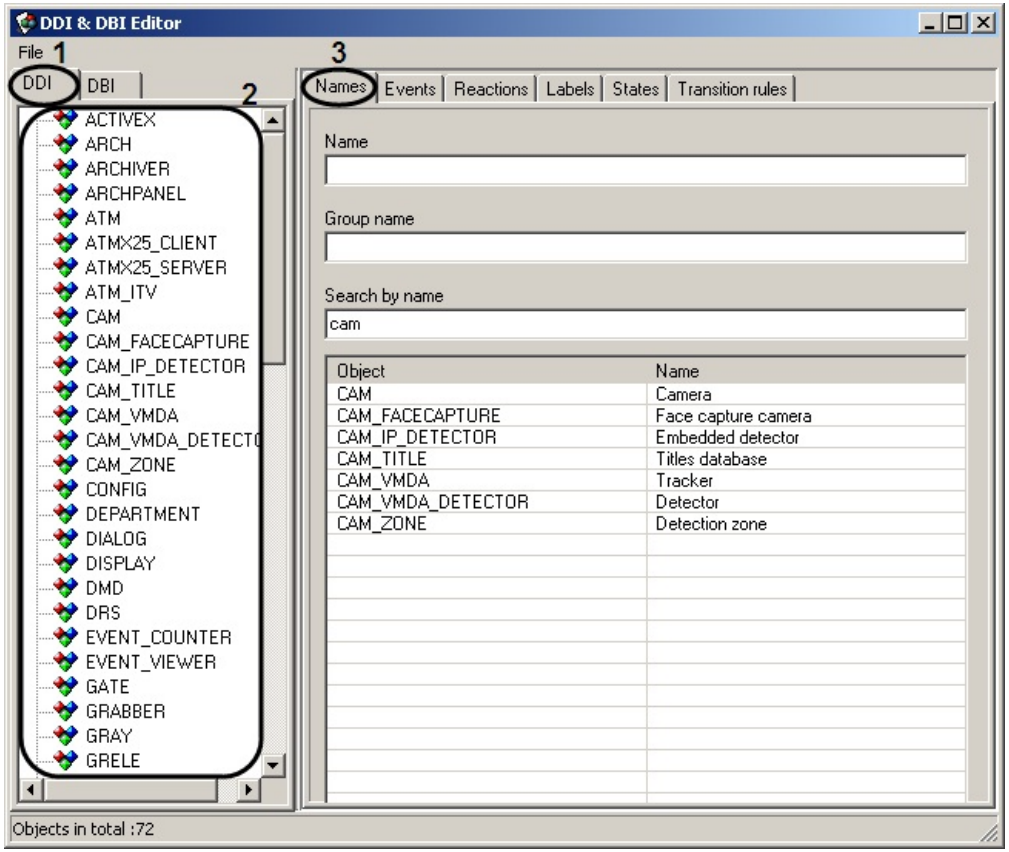
   ddi.exe tool is available in the **Start** menu with the following types of Intellect installation: Server, Administrator's workplace and Workplace for monitoring

2. In the **Tools** folder of Intellect installation folder.

To view the list of object names, events and reactions, do the following:

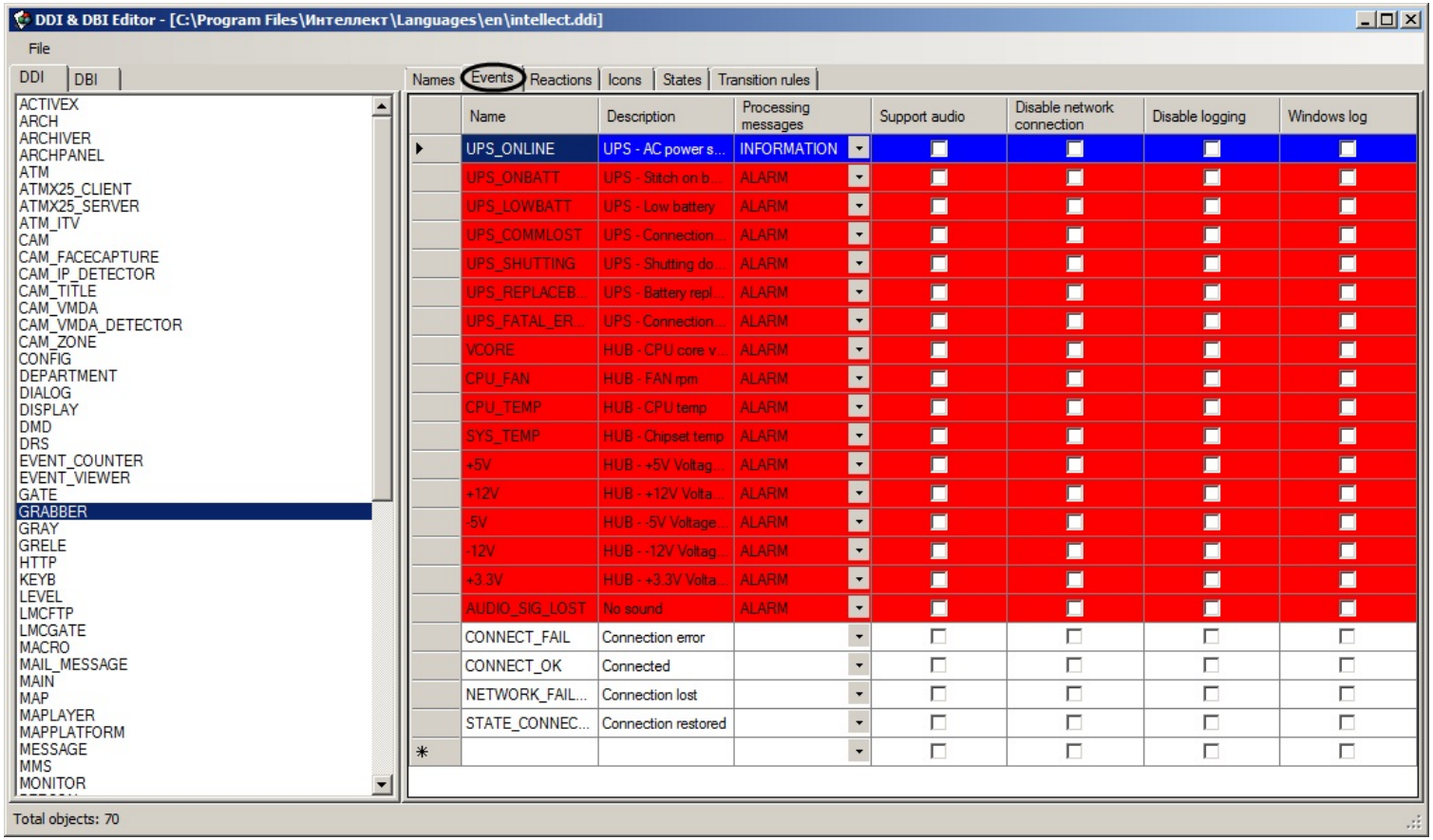
1. In the tool, open *intellect.ddi* file.

2. Select the **DDI** tab on the left (**1**). The list of system objects is displayed here.

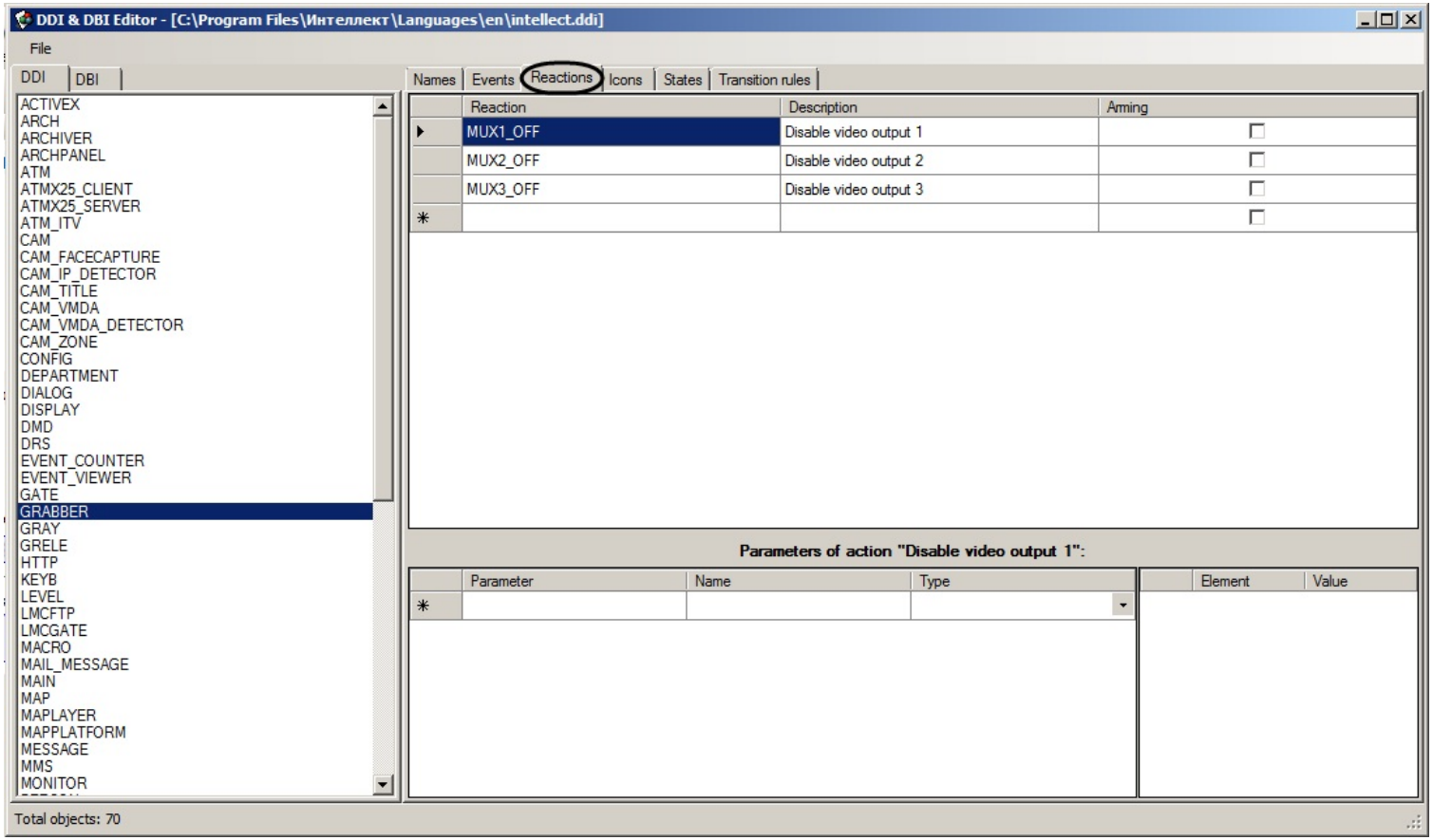


3. In the **DDI** tab, select the object which events and reactions are to be viewed (**2**).
4. To view the name of the selected object, go to the **Names** tab (**3**).

5. To view the list of events for the selected object, go to the **Events** tab.



6. To view the list of reactions for the selected object, go to the **Reactions** tab.



The detailed information on working with ddi.exe tool is given in Administrator's guide.

> **Note.**
>
> If the Sensor is armed, then at the Sensor switching on/off the "Alarm" event appears depending on the alarm mode setting (see the Creating and configuring the Sensor system object section of the Installing and configuring security system components guide). If the Sensor is disarmed, the "Closed" / "Opened" events appear correspondingly.

Operations with scripts

Creating a script

To create a JScript 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

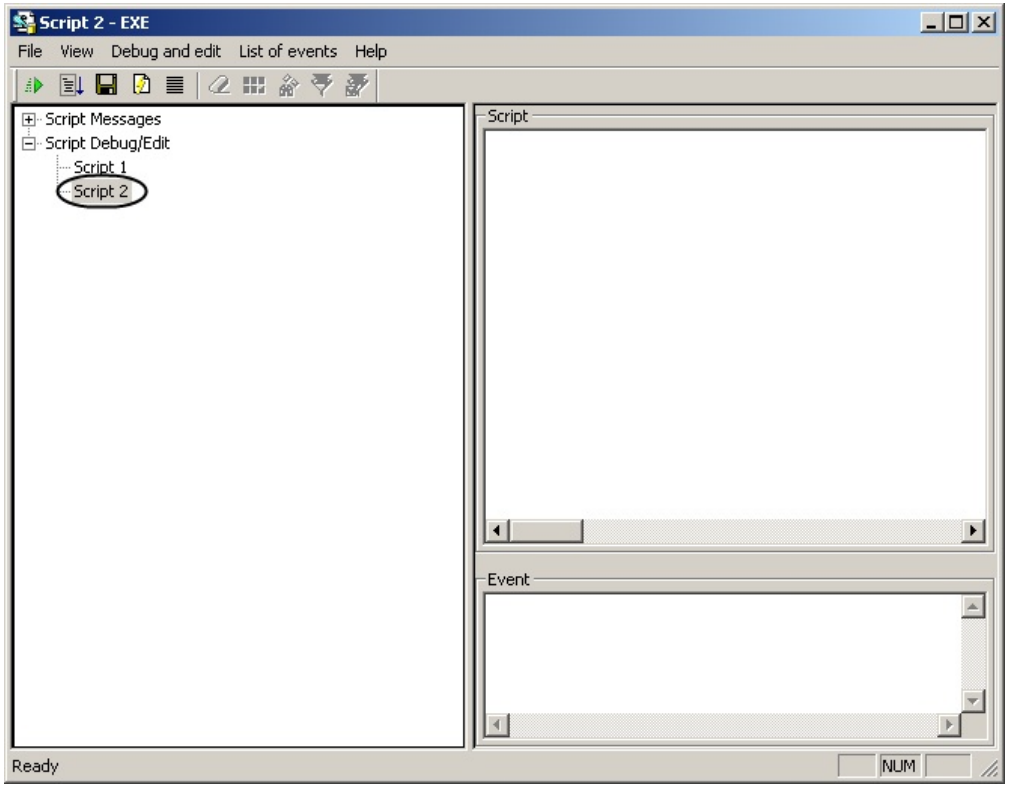
3. In the **Computers** field, select the computers (kernels) where the script should run.

   > **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.

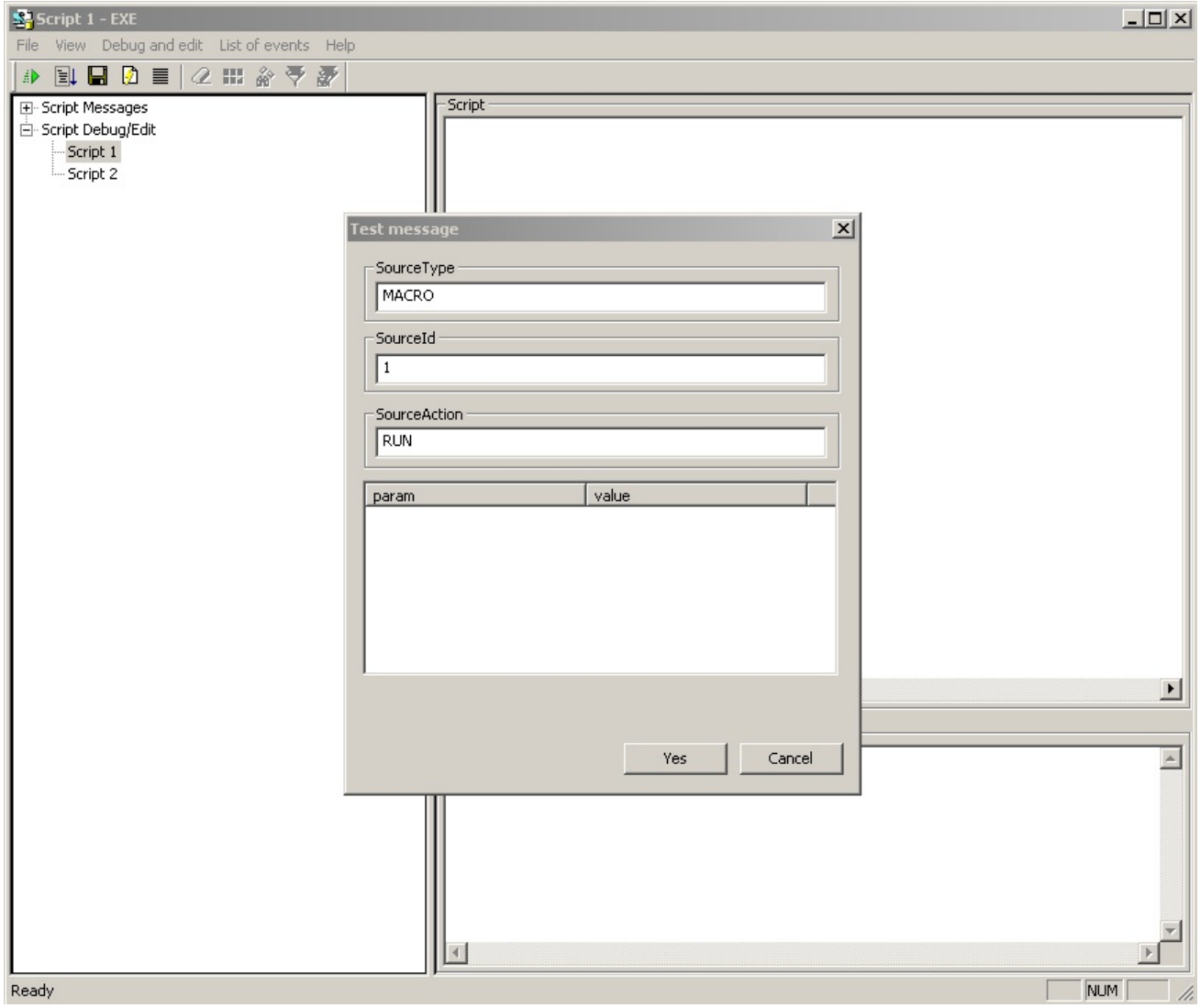
   > **Note**
   >
   > Use the **Editor-Debugger** utility to create, edit and save scripts in JScript. The text of saved script available to be edited is saved on the setting panel of the **Script** object.

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).



6. Enter the text of the script in JScript 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.



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

> **Note.**
>
> Also, use **Ctrl+T** hotkey to test-run the script by test event.

9. 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

10. 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.

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 **File** menu, select **Save in the database**.

> **Note.**
>
> Also, use **Ctrl+S** hotkey to save the script 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**.

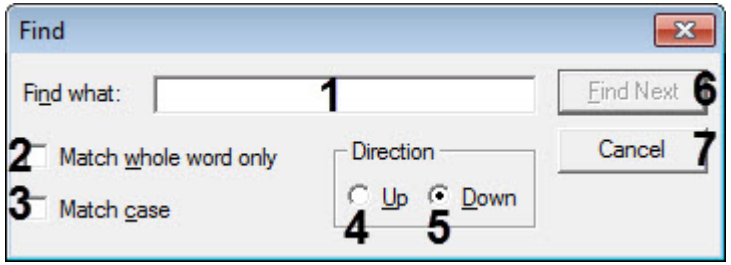
Deleting a script

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

Search text in script

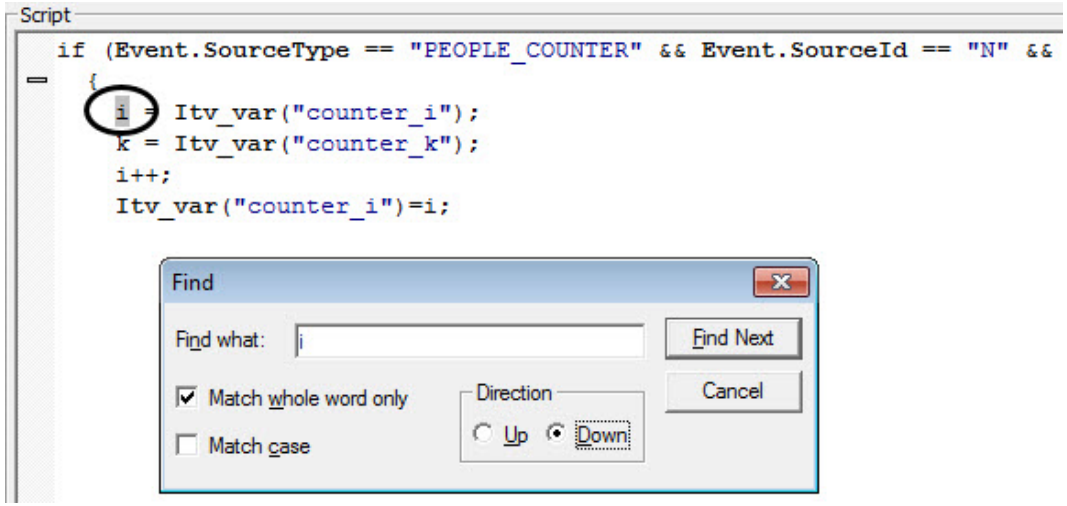
The *Editor-Debugger* allows searching text in script using a dialog box.

Press **Ctrl+F** to open search dialog box. The **Find** dialog box opens.



1. Enter the text to find in the **Find what:** field (1).
2. Set the **Match whole word only** checkbox to search for the text in whole (2).
3. Set the **Match case** checkbox to case-sensitive search for the text (3).
4. Select the direction of search through the text relatively to the current cursor position: **Up** (4) or **Down** (5).
5. Click **Find Next** to run search or go to the next entry (6).
6. Click **Cancel** to stop search and close the dialog box.

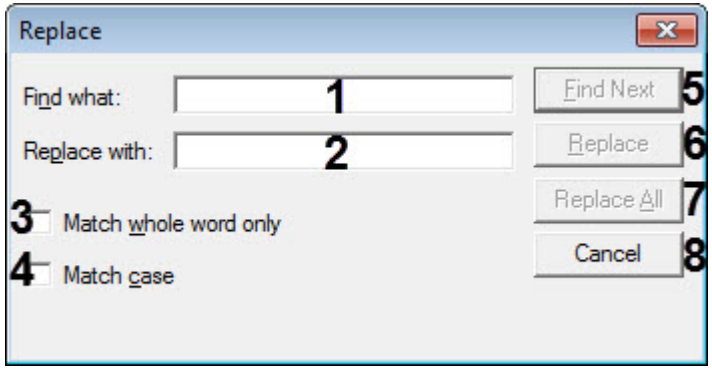
The found match is highlighted in the *Editor-Debugger* utility window.



Replace text in script

The *Editor-Debugger* allows replacing text in script using a dialog box.

Press **Ctrl+H** to open replace dialog box. The **Replace** dialog box opens.



1. Enter the text to find in the **Find what:** field (1).
2. Enter the text to replace the found text with in the **Replace with:** field (2).
3. Set the **Match whole word only** checkbox to search for the text in whole (3).
4. Set the **Match case** checkbox to case-sensitive search for the text (4).
5. Click **Find Next** to run search or go to the next entry (5).

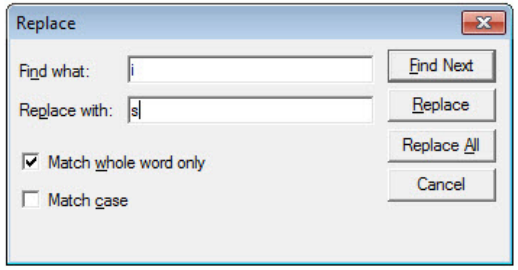
> **Note.**
>
> The search runs down from the current cursor position.

6. Click **Replace** to replace the current found match (6).
7. Click **Replace All** to replace all matches automatically (7).
8. Click **Cancel** to stop close the replace dialog box (8).

Example of use fo replacing variable **i** to **s**:

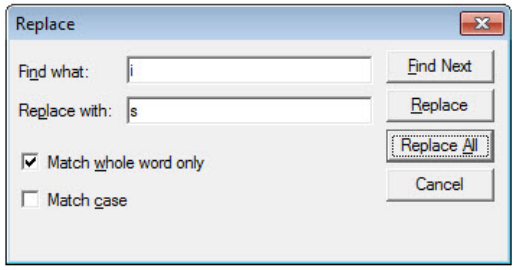
1. Before replacement:

```
Script
if (Event.SourceType == "PEOPLE_COUNTER" && Event.SourceId == "N" &&
  {
    i = Itv_var("counter_i");
    k = Itv_var("counter_k");
    i++;
    Itv_var("counter_i")=i;
```



2. After replacement:

```
Script
if (Event.SourceType == "PEOPLE_COUNTER" && Event.SourceId == "N" &&
  {
    s = Itv_var("counter_i");
    k = Itv_var("counter_k");
    s++;
    Itv_var("counter_i")=s;
```

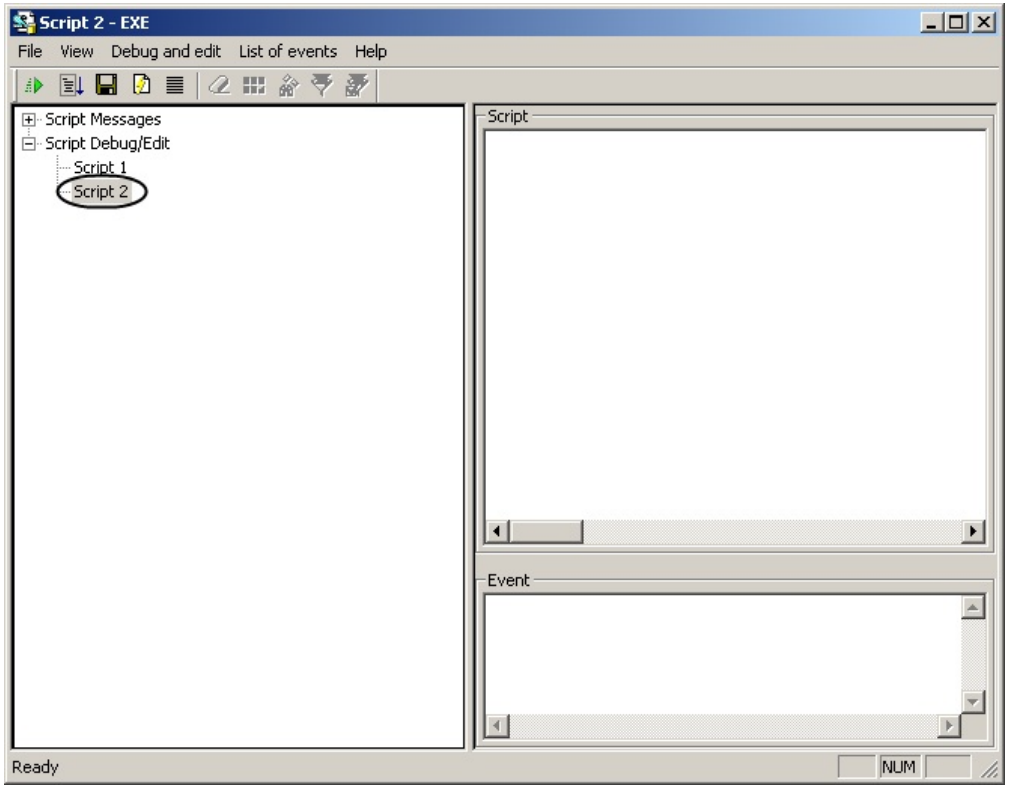


Creating your first script

As an example of using JScript 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. The Events table should not be filled for .
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.

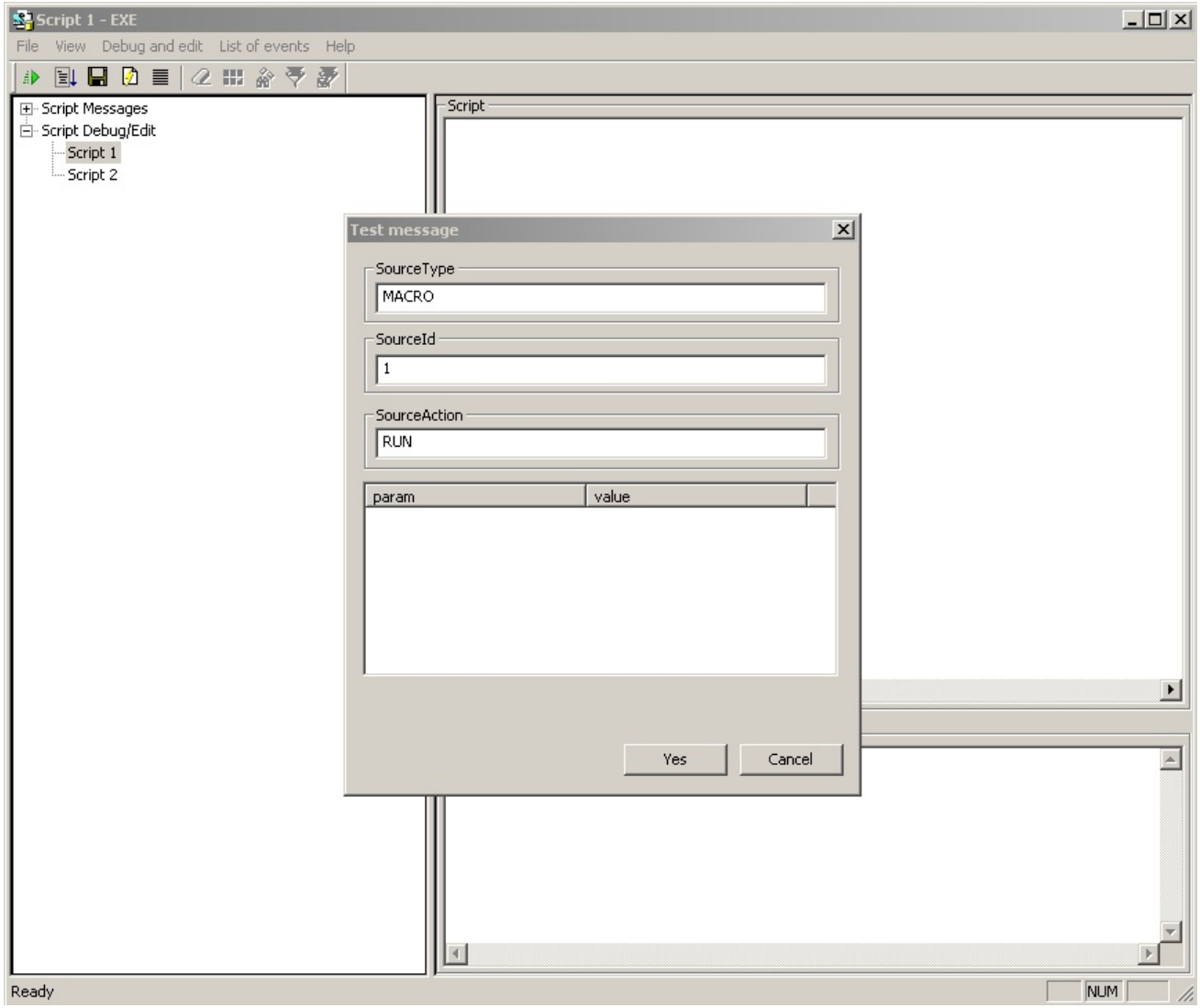


7. Enter the following in the **Script** field:

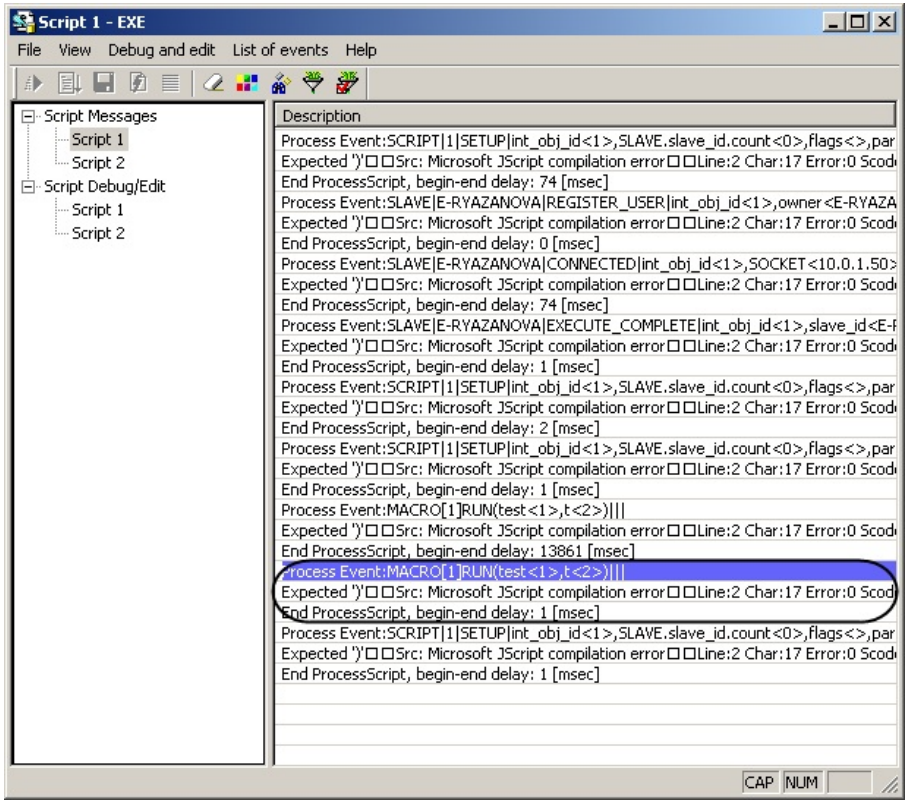
```
 if (Event.SourceType == "MACRO" && Event.SourceId == "1" &&
Event.Action == "RUN")
{
  var ;
  for(i=1; i<=4; i=i+1)
  {
 SetObjectParam("CAM",i,"hot_rec_time","10");
  }
```

```
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.



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".



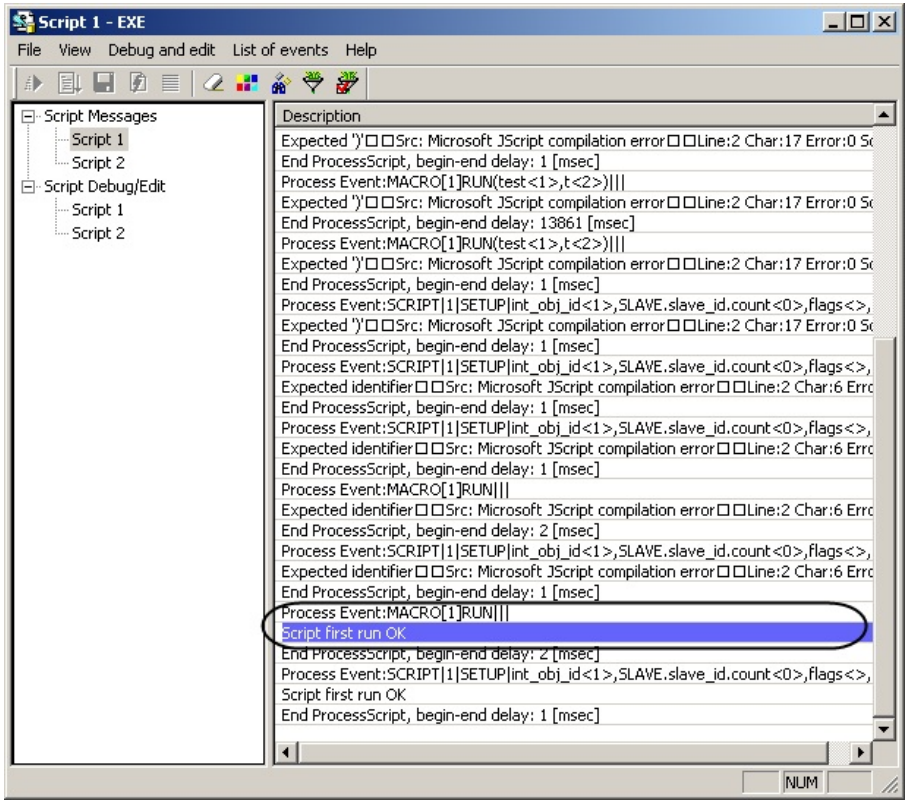
The error message says that there is no identifier in the second line of variable declaration operator (var). That means no variable has been declared. This is a critical error in JScript, thus the script has not been executed.

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");
}
```

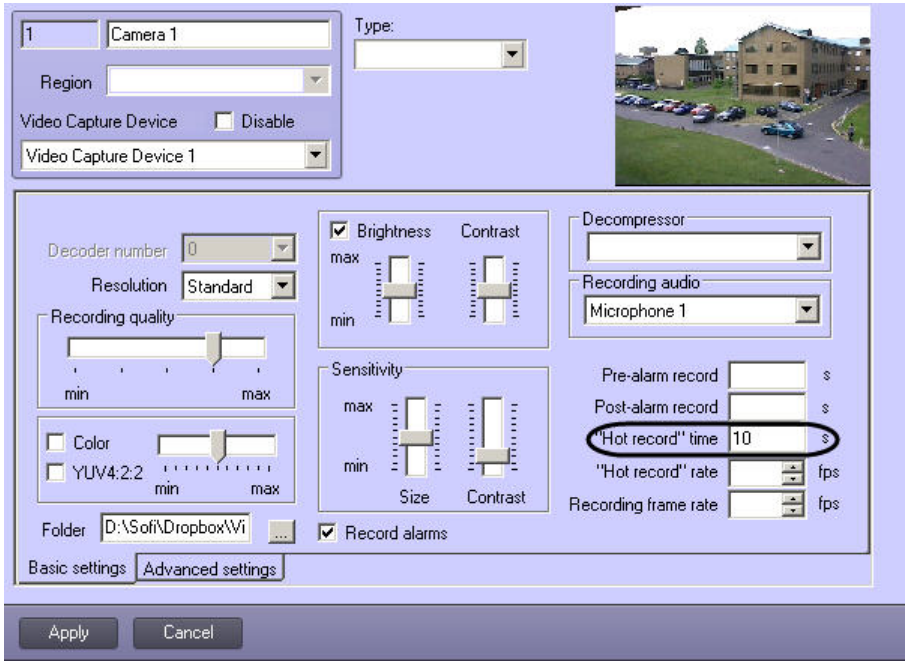
14. In the **File** menu, select **Save to database** to save the script.
15. Repeat steps 10 and 11.
16. In the debugger window, find the "Process Event:MACRO|1|RUN|" line and the "DebugLogString:Hello world" and "Script first run OK" messages. The "Script first run OK" means that the script runs correctly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



17. Close the *Editor-Debugger* utility.
18. The text of the created script will be shown in the **Script f**ield 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.

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”.



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

Script creation and debugging is now complete.

Script debugging

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.

Creating and using test events

Creating test events

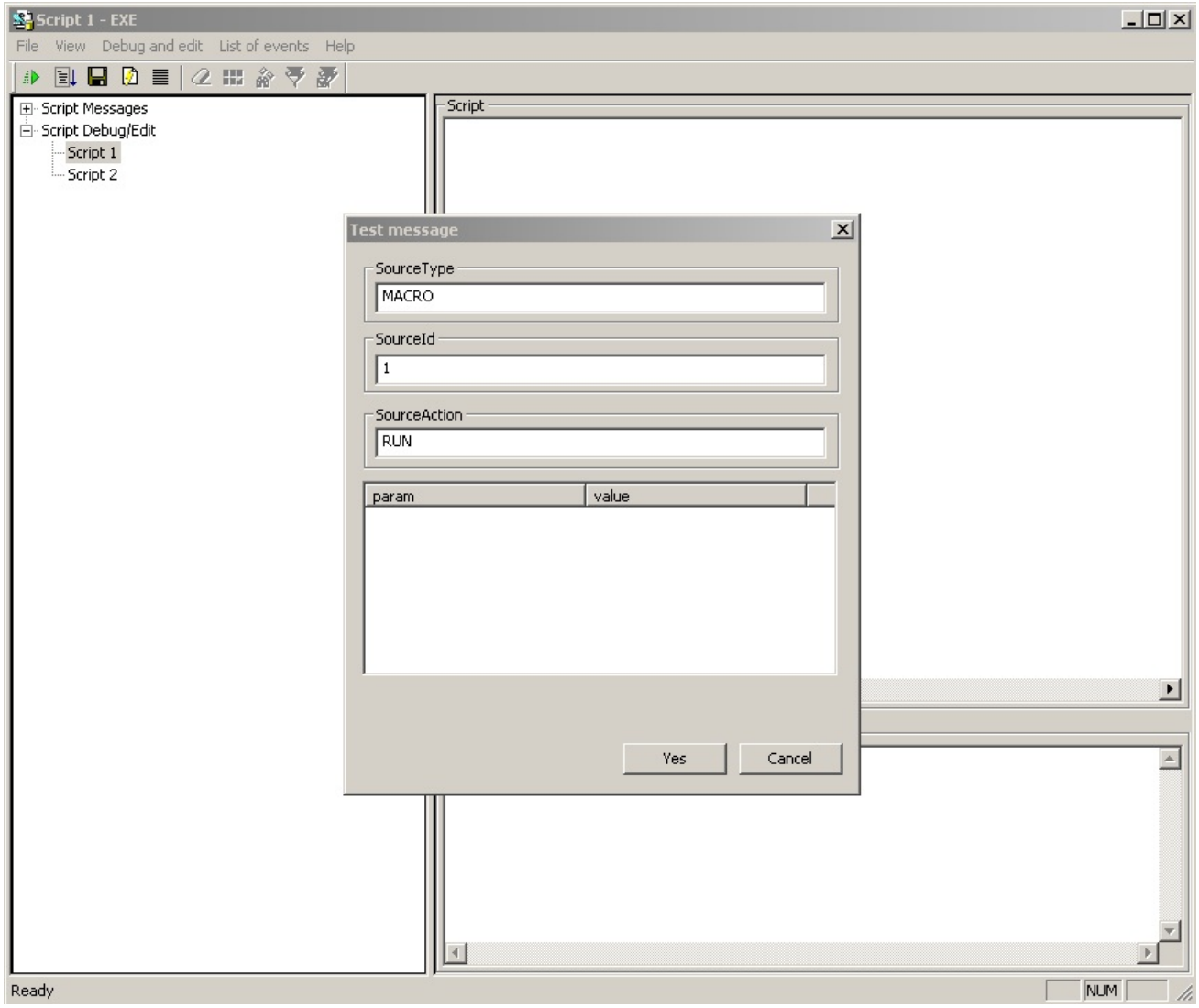
The *Editor-Debugger* utility is capable of generating test events chosen by the user to help debug the scripts. Test events are not registered by the video surveillance system, i. e. they are not listed in the events log and not saved to the database.

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.



This window is used for entering the test event parameters.

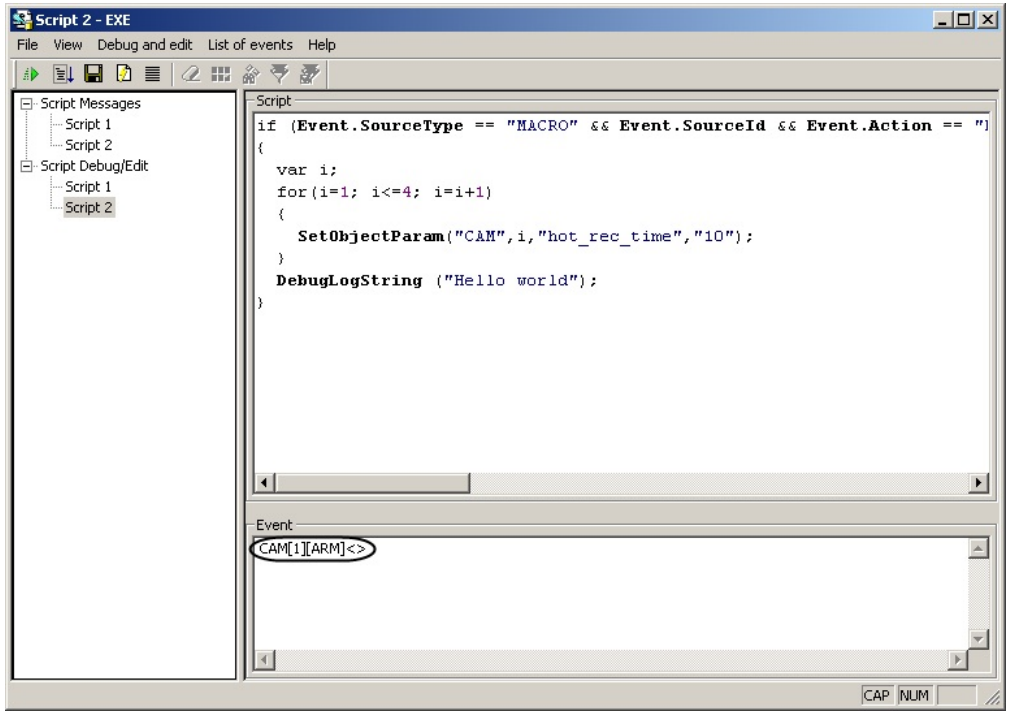
3. Enter the following information in the fields of the Test message window:
    1. **SourceType** – object type;
    2. **SourceId** – object identification number;
    3. **SourceAction** – the event generated by the specified object;
    4. **param** – additional event parameters;
    5. **value** – 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 shows the test event example: “Arm Camera 111”.



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.

Using debugger windows of the Editor-Debugger utility

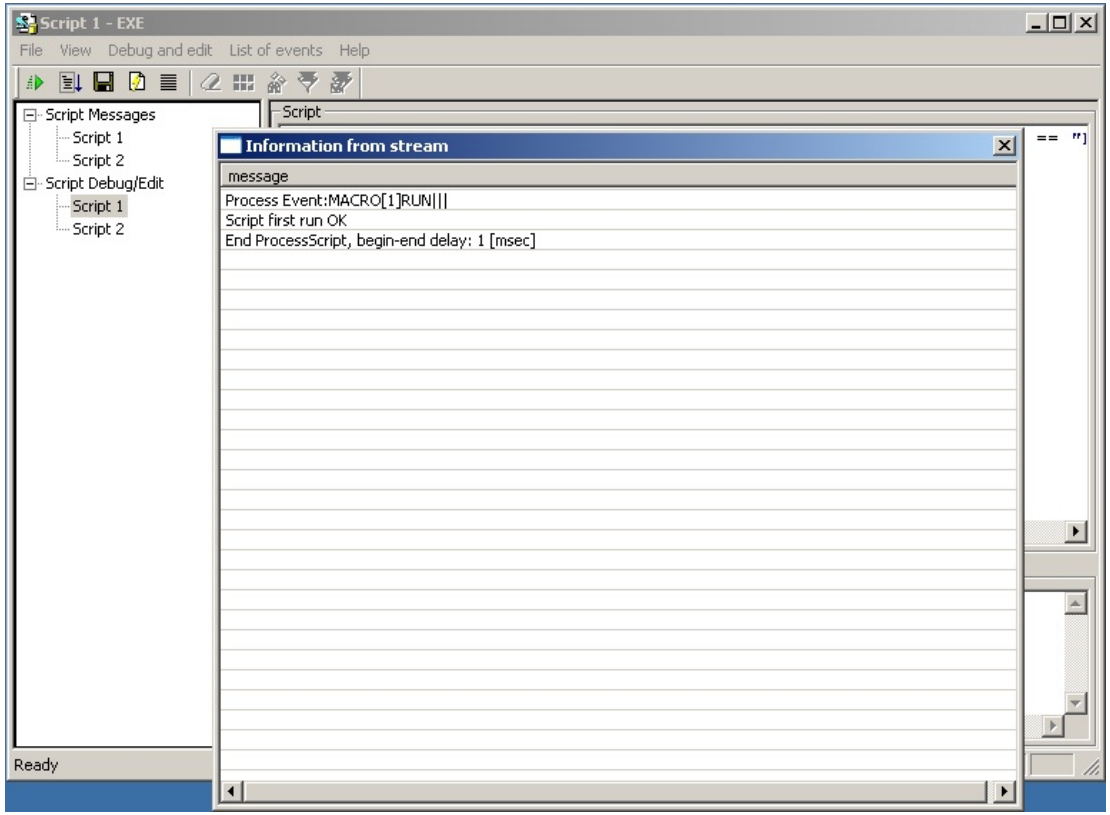
Debugger window types: Script Messages and Thread Information

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

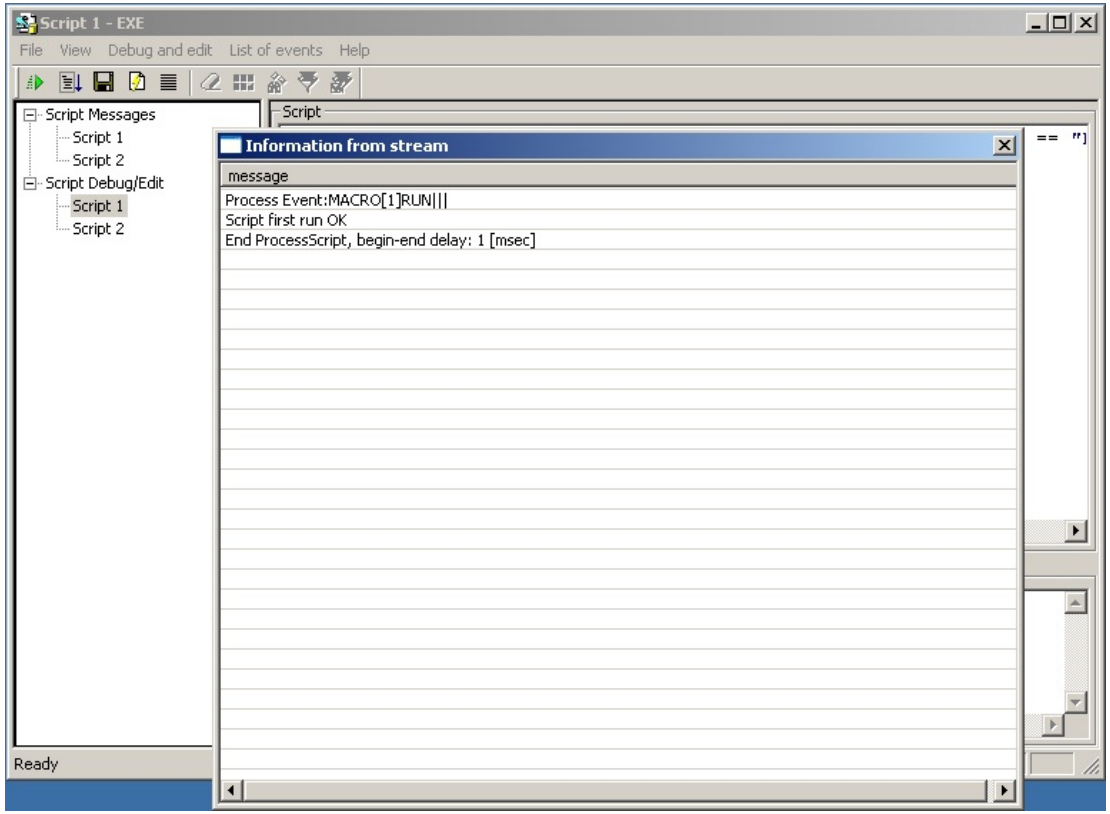
A separate debugger window is assigned to each script in the *Editor-Debugger* utility.

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 I*ntellect* system.



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.



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

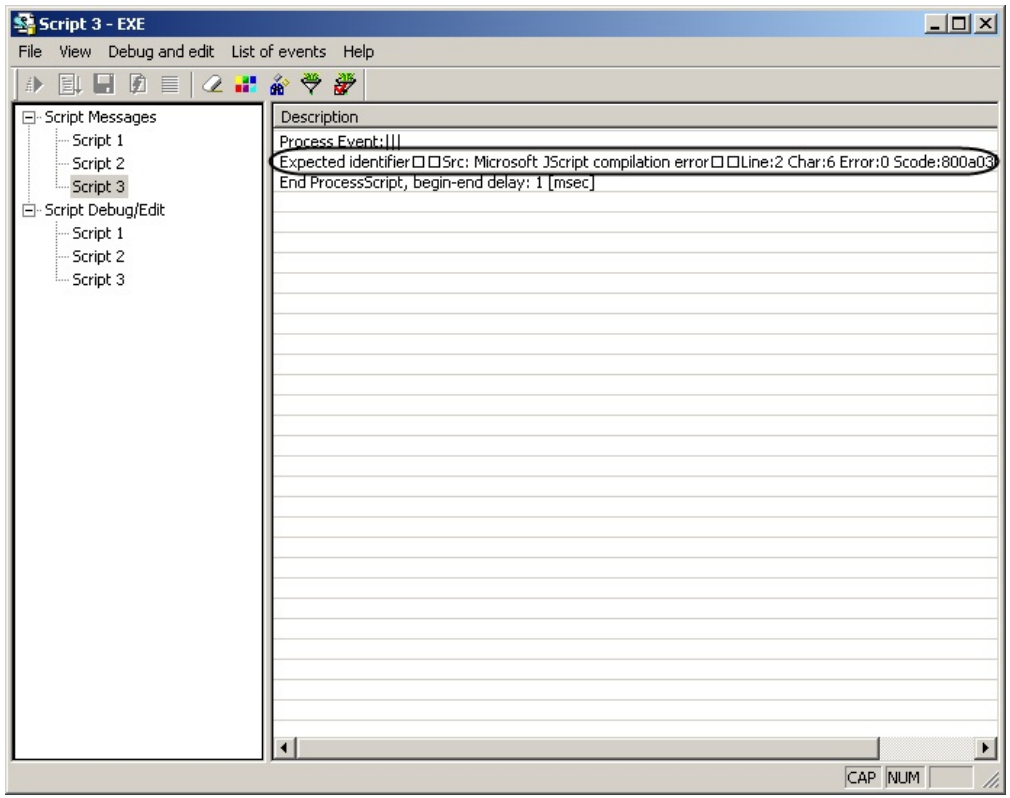
Displaying messages about starting, verifying, changing and executing scripts in the debugger windows

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

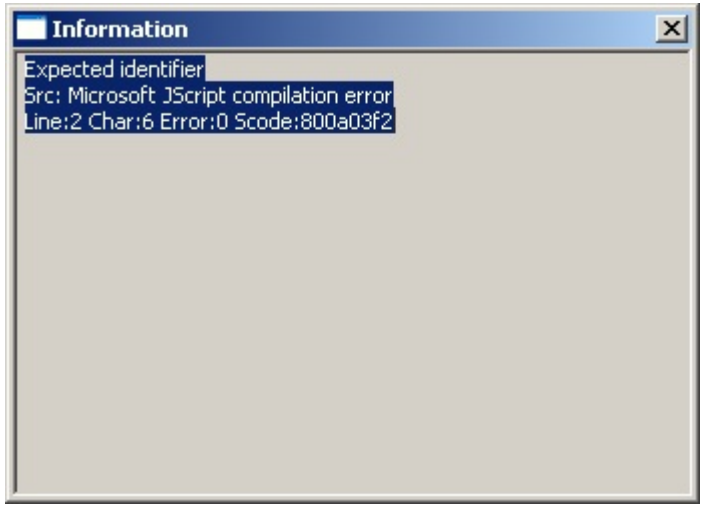
When the event occurs that triggers the script, the following message is displayed in the debugger window: “Process Event: <script triggering event>”; for example, if the script starts upon Macro 1 execution, the line reads “Process Event:MACRO|1|RUN|”.

At the moment of changing the script in *Debugger* utility or in the Intellect, the debugger window displays the message «Process Event: SCRIPT|script’s number|SETUP|» (for example, while changing the script with number 1, the debugger window displays «Process Event: SCRIPT|1|SETUP|»).

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



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]".

Using third-party debugger programs

The *Intellect* software package officially supports debugger *Microsoft Visual Studio 2005*.

The *Intellect* software package allows using third-party debuggers for processing JScript 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.

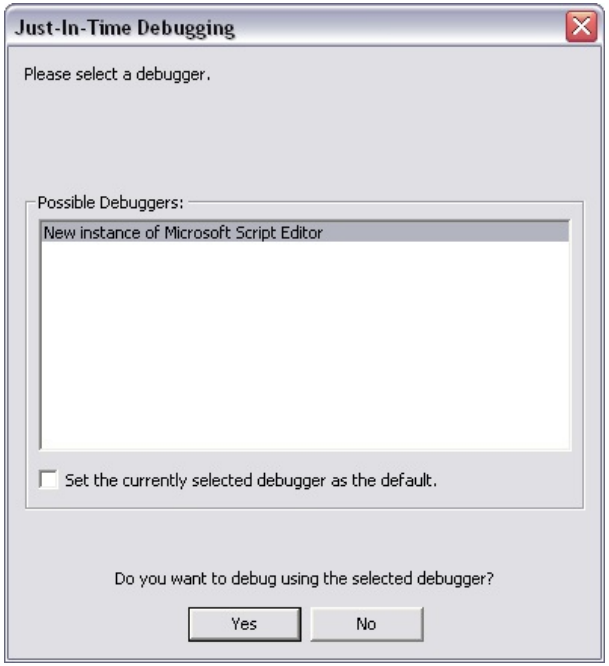
> **Note**
>
> We do not recommend using third-party debuggers, since they do not provide full compatibility with the Intellect software, and may lead to failure of the *Intellect* software

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.



5. 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 when Macro 1 starts.

```
 if (Event.SourceType== "MACRO" && Event.SourceId=="1" && Event.Action == "RUN"); //start Macro 1
{
debugger; // breakpoint
DebugLogString ("Hello world");
}
```

Examples of scripts in JScript language

On the page:

To illustrate available fields of application of scripts in Jscript see the following examples which can be used to create additional functions in the system on the basis of the **Script** object.

Example 1. Visualisation of operating the Queue length detection in the Video Surveillance monitor

The **Queue length detection** object (is part of the Detector Pack package), **Camera** and **Captioner** objects (instead of N, M, L symbols set the corresponding numbers of the Queue length detection, Camera and Captioner objects) are to be created and configured for correct script working in the *Intellect* software.

```
//Event reading by the queue length
if (Event.SourceType == "OCCUPANCY_COUNTER" && Event.SourceId == "N" && Event.Action == "OCCUPANCY")  //N -
Number of Queue length detection
{
  var n=Event.GetParam("occupancy");
//Displaying the queue number by the Captioner in the Monitor
  DoReactStr("CAM","M","CLEAR_SUBTITLES","title_id<L>"); //M - Number of Camera  L - Number of Captioner
  DoReactStr("CAM","M","ADD_SUBTITLES","command<Queue length: "+n+" person(s).\r>,page<BEGIN>,title_id<L>");
//M, L - the same
}
```

As a result the text message about the current queue length will be imposed on the video image while displaying the corresponding camera in the Video Surveillance monitor.

Settings of the font, color and position of text is configured on the settings panel of the **Captioner** object.

Example 2. Visualisation of operating the People Counter detection in the Video Surveillance monitor

The **People Counter Detection** object (is part of the Detector Pack package), **Camera**, **Captioner** and **Macro** objects (instead of N, M, L, P symbols set the corresponding numbers of the People counter detection, Camera, Captioner and Macro objects) are to be created and configured for correct script working in the *Intellect* software.

```
//Event reading and counting of entered people
if (Event.SourceType == "PEOPLE_COUNTER" && Event.SourceId == "N" && Event.Action == "IN") //N - Number of
People Counter detection
  {
    i = Itv_var("counter_i");
    k = Itv_var("counter_k");
    i++;
    Itv_var("counter_i")=i;
//Displaying the number of people by Captioner in the Monitor

    DoReactStr("CAM","M","CLEAR_SUBTITLES","title_id<L>"); //M - Number of Camera  L - Number of Captioner
    DoReactStr("CAM","M","ADD_SUBTITLES","command<Number of people (entering/exiting): "+i+" / "+k+"\r>,
page<BEGIN>,title_id<L>"); //M, L - the same

  }
//Event reading and counting of exitng people
if (Event.SourceType == "PEOPLE_COUNTER" && Event.SourceId == "N" && Event.Action == "OUT") //N - Number of
People Counter detection
  {
    i = Itv_var("counter_i");
    k = Itv_var("counter_k");
    k++;
    Itv_var("counter_k")=k;
//Displaying the number of people by Captioner in the Monitor

   DoReactStr("CAM","M","CLEAR_SUBTITLES","title_id<L>"); //M - Number of Camera  L - Number of Captioner
   DoReactStr("CAM","M","ADD_SUBTITLES","command<Number of people (entering/exiting): "+i+" / "+k+"\r>,
page<BEGIN>,title_id<L>"); //M, L - the same
```

```
  }
//Null the counter by Macro (previously the Macro is to be created in the Intellect)
if (Event.SourceType == "MACRO" && Event.SourceId == "P" && Event.Action == "RUN") //P - Number of Macro
  {
    Itv_var("counter_i")=0;
    Itv_var("counter_k")=0;
    i=0;
    k=0;
//Displaying number of people by Captioner in the Monitor

    DoReactStr("CAM","M","CLEAR_SUBTITLES","title_id<L>"); //M - Number of Camera  L - Number of Captioner
    DoReactStr("CAM","M","ADD_SUBTITLES","command<Number of people (entering/exiting): "+i+" / "+k+"\r>,
page<BEGIN>,title_id<L>"); //M, L - the same
  }
```

As a result the text message about number of entering and exiting people will be imposed on the video image while displaying the corresponding camera in the Video Surveillance monitor.

Settings of the font, color and position of text is configured on the settings panel of the **Captioner** object (see the Configuring captions display on a video image section of the Administrator's Guide document).

Previously create the **Macro** object on the **Programming** tab to null the counter of visitors. The name of the object can be changed to "Null of people counter" for ease of using.

Macro of setting to zero can be run manually by main menu of the *Intellect* software package or automatically in any specified time (use the **Events** table on the settings panel of the **Macro** object where the previously configured **Time zone** object is to be specified). Detailed information about using the **Macro** and **Time zone** objects is given in the Administrator's Guide document).

Example 3. Setting the text for displaying on the map

It is possible to select the type of **Text** displaying while attaching object to the map (see Administrator Guide, the Attaching objects to the layers of interactive map section). Also the script on the JScript language can be used for changing the displayed text.

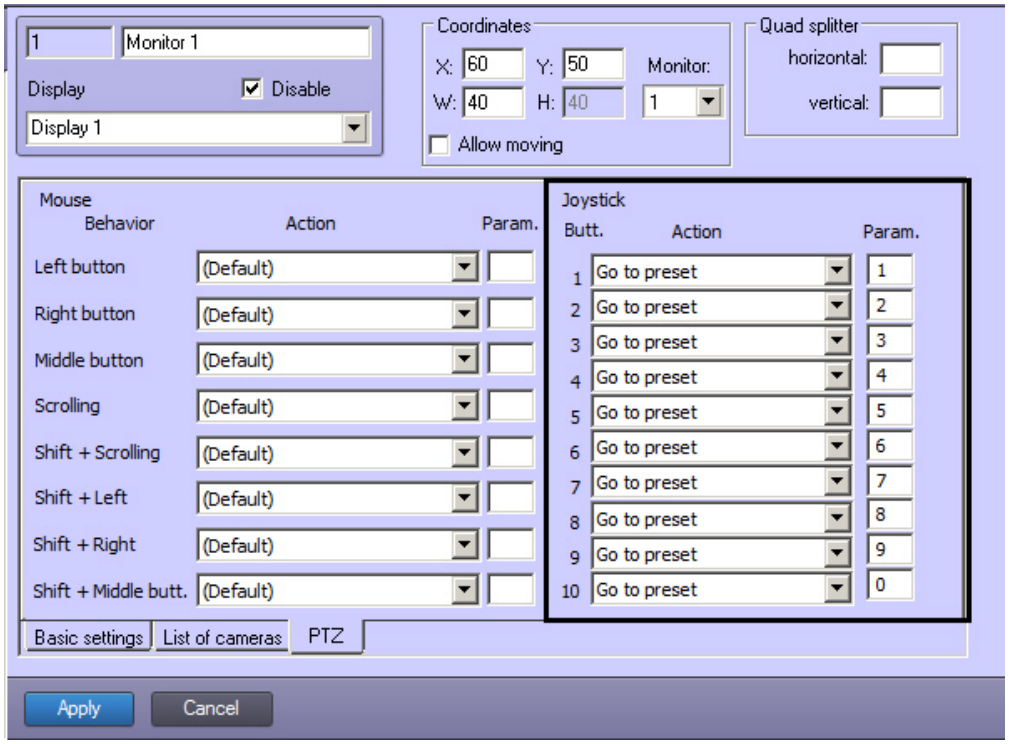
Example. The Text type of display on the map is selected for the Camera 1. It's required to display value of the MyVar value read from the C:\test.ini file on the map and debug window while processing the Macro 1.

```
if(Event.SourceType == "MACRO" && Event.Action == "RUN")
{
 var result = parseInt(ReadIni("MyVar","C:\\test.ini"));
 result += 2;
 NotifyEventStr("CAM","1","ANALOG_PARAMS","text<Variable value = \n" + result + ">, blink_state<1>");
```

```
  DebugLogString(result);
}
```

Example 4. Displaying camera on the monitor by clicking the button on the remote control

The following example is valid only for cameras in configuration of which there is PTZ control panel. When configuring Video surveillance monitor select the **Go to preset** action with 1,2,3...,0 parameters for ten joystick buttons (see Assigning commands to joystick buttons using the Monitor section of Installing and configuring security system components guide).



Example. When the button is clicked on the control panel, display corresponding camera in the active monitor. The script is to timer trigger with ID=1.

> **Note.**
>
> The **Timer** object is to be created and configured beforehand and the current year is to be set. Information on how to configure the Timer object can be found in Creating and configuring the Timer object section of Administrator's Guide

After each button click on the control panel wait for 2 seconds until clicking another button. If there is no button click, then the camera with dialed number is to be displayed.

```
if (Event.SourceType=="TIMER" && Event.SourceId=="1" && Event.Action=="TRIGGER")
{
  mon="1";
  DebugLogString("on monitor "+ Itv_var("cam"));
  DoReactStr("MONITOR",mon,"ACTIVATE_CAM","cam<"+Itv_var("cam")+">");
  Itv_var("cam")="";
}

if (Event.GetParam("source_type")=="TELEMETRY" && Event.GetParam("action")=="GO_PRESET")
{
  DoReactStr("TIMER","1","START","bound<2>");
  var key=Event.GetParam("param4_val");
  DebugLogString("Key:"+key);
  Itv_var("cam")=Itv_var("cam")+key;
  DebugLogString(Itv_var("cam"));
}
```

Example 5. Sending command to camera using the camera HTTPAPI

Example. Camera IP address is 192.168.0.13.

The command enables the camera wiper:

`192.168.10.101/httpapi/SendPTZ?action=sendptz&PTZ_PRESETSET=85`

The command disables the camera wiper:

`192.168.10.101/httpapi/SendPTZ?action=sendptz&PTZ_PRESETSET=86`

These commands are to be sent to camera using the Jscript script.

```
function DoPreset(preset)
{
    xmlhttp=new ActiveXObject("MSXML2.XMLHTTP");
    if(xmlhttp == null)
    {
```

```
            return;
    }
    xmlhttp.open("GET", "http://192.168.0.13/httpapi/SendPTZ?action=sendptz&PTZ_PRESETSET="+preset, false,"
admin","1234");

    xmlhttp.send();
    DebugLogString(xmlhttp.status);
}
if (Event.SourceType == "MACRO" && Event.SourceId == "6" && Event.Action == "RUN")
{
 DoPreset("85");
}

if (Event.SourceType == "MACRO" && Event.SourceId == "7" && Event.Action == "RUN")
{
 DoPreset("86");
}
```

Example 6. Script for framing abandoned objects in live video

If the object tracking function is in use (see Creating and configuring the Timer object section of Administrator's Guide), then detected objects will be framed in video when viewing the archive. To frame abandoned objects in live video use the script aimed to frame an abandoned object when receiving alarm from VMDA detection tool:

```
if (Event.SourceType=="CAM_VMDA_DETECTOR")
{
 cam=GetObjectParentId("CAM_VMDA_DETECTOR",Event.SourceId,"CAM");
 if (Event.Action=="ALARM")
 {
   var x1,x2,y1,y2;
   x1=Event.GetParam("x");
   x2=Event.GetParam("w");
   y1=Event.GetParam("y");
   y2=Event.GetParam("h");
   x2=parseInt(x1)+parseInt(x2);
```

```
   y2=parseInt(y1)+parseInt(y2);
   DoReactStr("MONITOR","","SET_MARKRECT","cam<"+cam+">,color<255>,id<"+cam+">,x1<"+x1+">,x2<"+x2+">,y1<"+y1+">,
y2<"+y2+">");
   DebugLogString("x1:"+x1+" x2:"+x2+" y1:"+y1+" y2:"+y2);
 }
 else
 {
   DoReactStr("MONITOR","","DEL_MARKRECT","cam<"+cam+">,id<"+cam+">");
 }
}
```

Example 7. Using START and STOP events for Failove

Objects from more than one main Server are not to be moved to the Backup Server. For this when moving objects from some main Server to the Backup Server all other **Failover** objects are to be disabled on this Backup Server.

```
if (Event.SourceType  == "FAILOVER" )
{
  if (Event.Action == "START") {action="DISABLE";}
  if (Event.Action == "STOP") {action="ENABLE";}
  id=Event.SourceId;
  msg=CreateMsg();
  msg.StringToMsg(GetObjectIds("FAILOVER"));
  var
  objCount=msg.GetParam("id.count");
    for (i=0;i<objCount;i++)
    {
     pid=msg.GetParam("id."+i);

     if (!(id==pid)) {
       DoReactStr("FAILOVER",pid,action,"");
     }
  }
}
```

Programming Guide (JScript). Conclusion

More detailed information on the Intellect software package is presented in the documents titled:

1. Installing and configuring security system components guide;
2. Operator's Guide;
3. Administrator's Guide.

If while operating the given software product you have faced difficulties and problems, you are welcome to contact us. However before addressing us, we kindly ask you to answer the following questions:

1. What is the problem?
2. When did the problem occurr and what had happened before it occurred?
3. Which conditions gave rise to the problem?

Remember, that the more detailed and precise information you give us, the faster our experts will resolve your problem.

We are striving to improve the quality of our products, and hence welcome any proposals and suggestions how to improve our software and documentation.

Please forward your suggestions to the following e-mail addresses: documentation@axxonsoft.com

Appendix 1. Description of the Editor-Debugger utility

The purpose of the Editor-Debugger utility

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

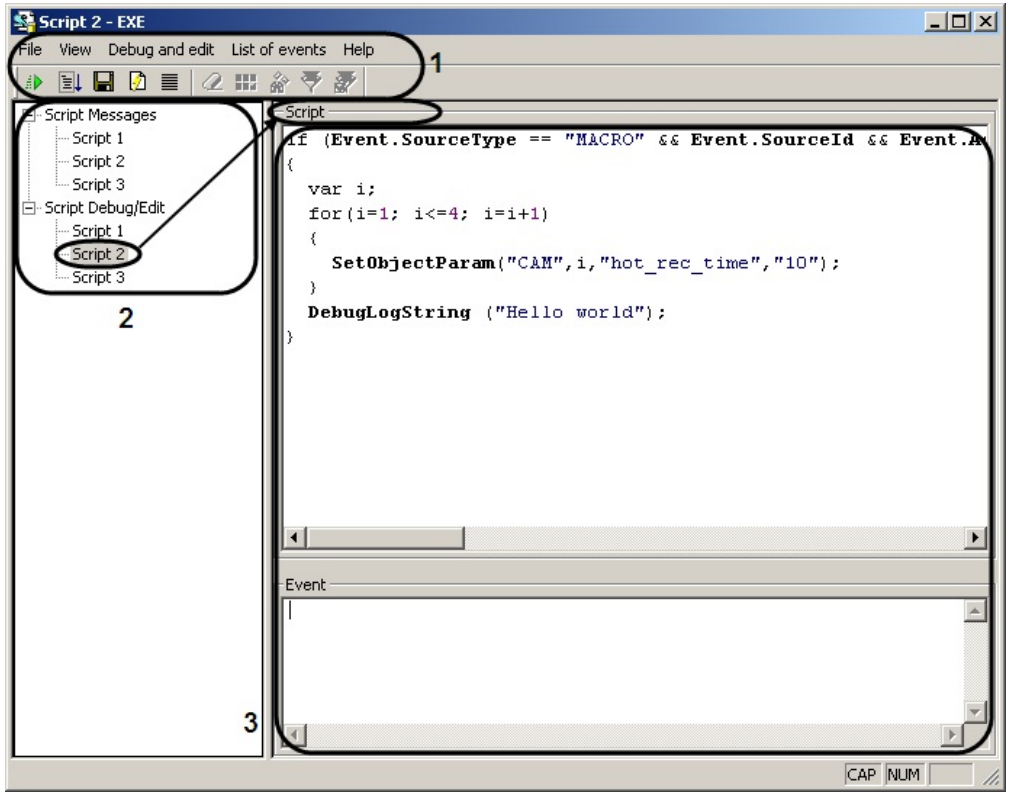
The *Editor-Debugger* utility provides the following functionality:

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.

The interface of the Editor-Debugger utility

The Editor-Debugger window

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



The Script Debug/Edit tab

Description of the Script Debug/Edit tab

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

Figure shows the **Script Debug/Edit** panel.

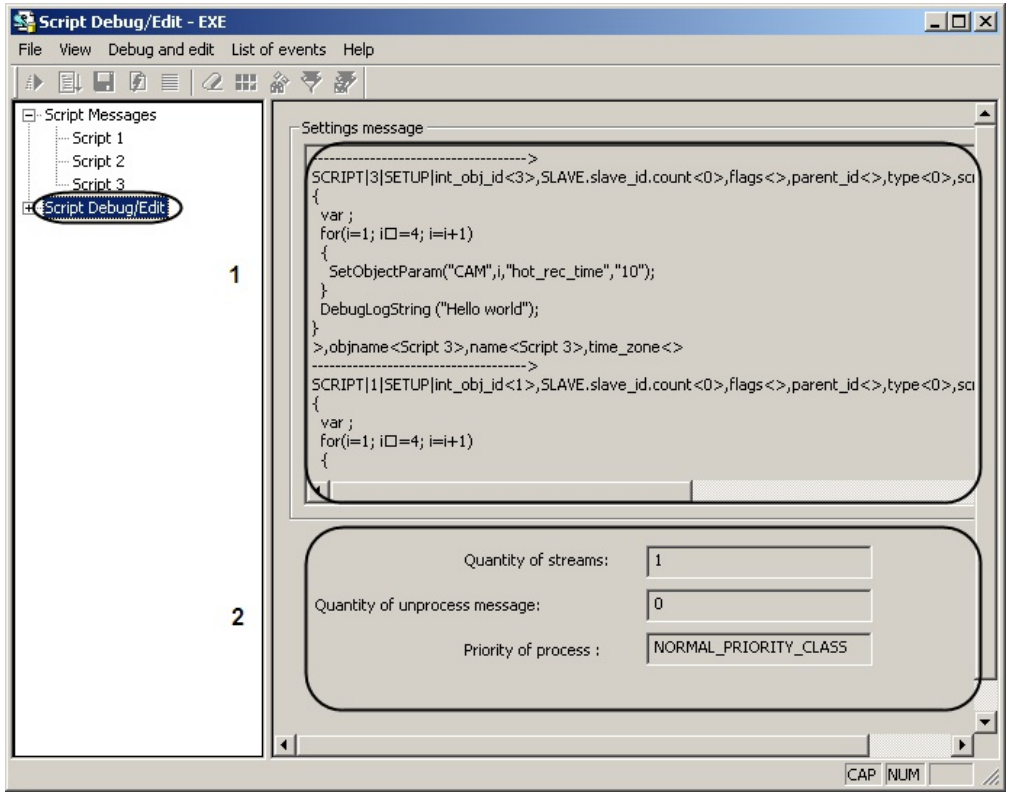


Table shows the description of the elements in the **Script Debug/Edit** panel.

№	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	-	-

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 shows the **Script** object panel.

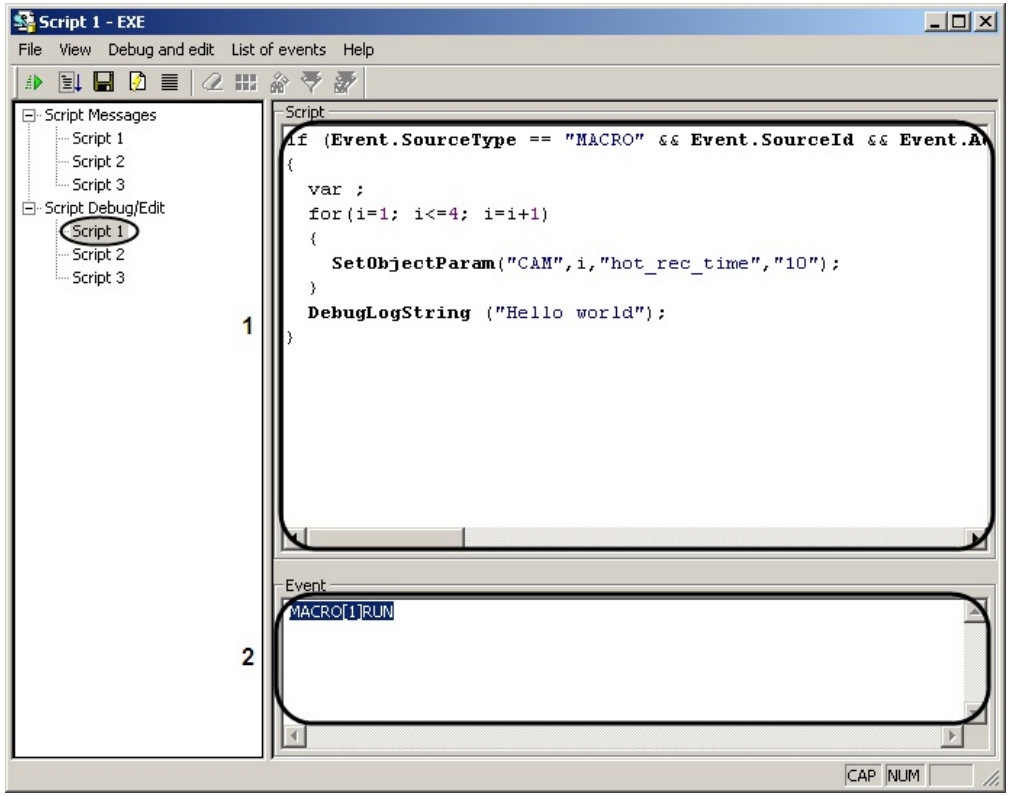


Table describes the elements in the **Script** object.

№	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

The Script Messages tab

Description of the Script Messages tab

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

Figure shows the **Script Messages** panel.

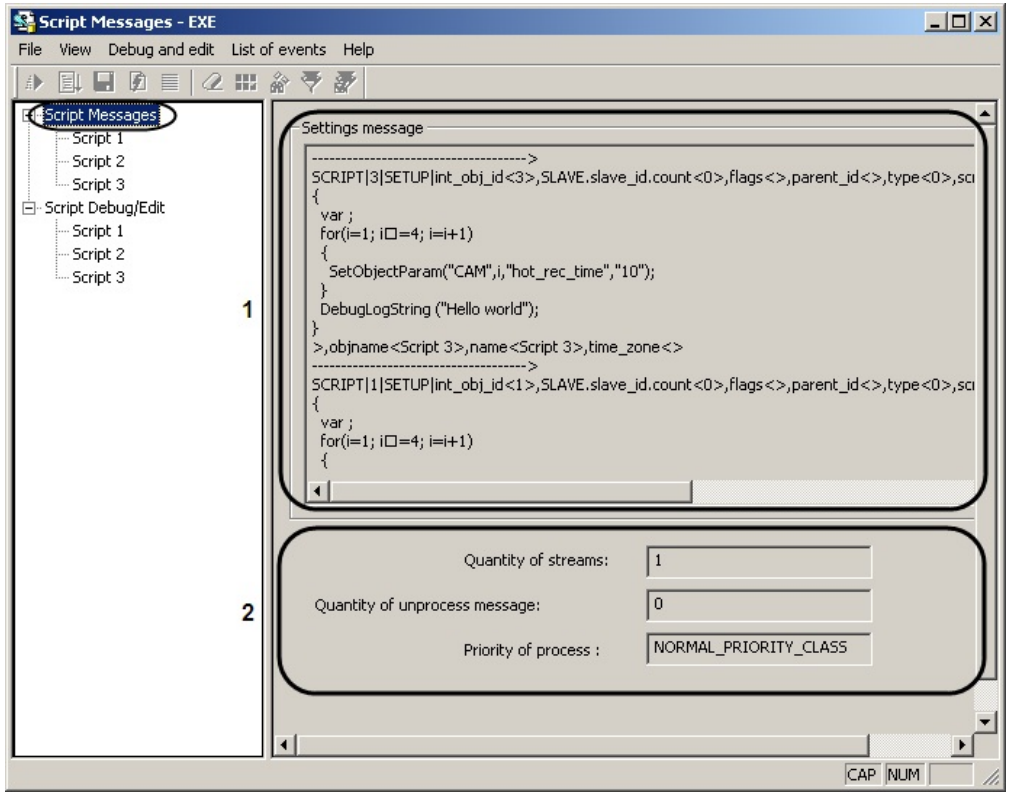


Table shows the description of the elements in the **Script Messages** panel.

№	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	-	-

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 shows the **Script** object panel.

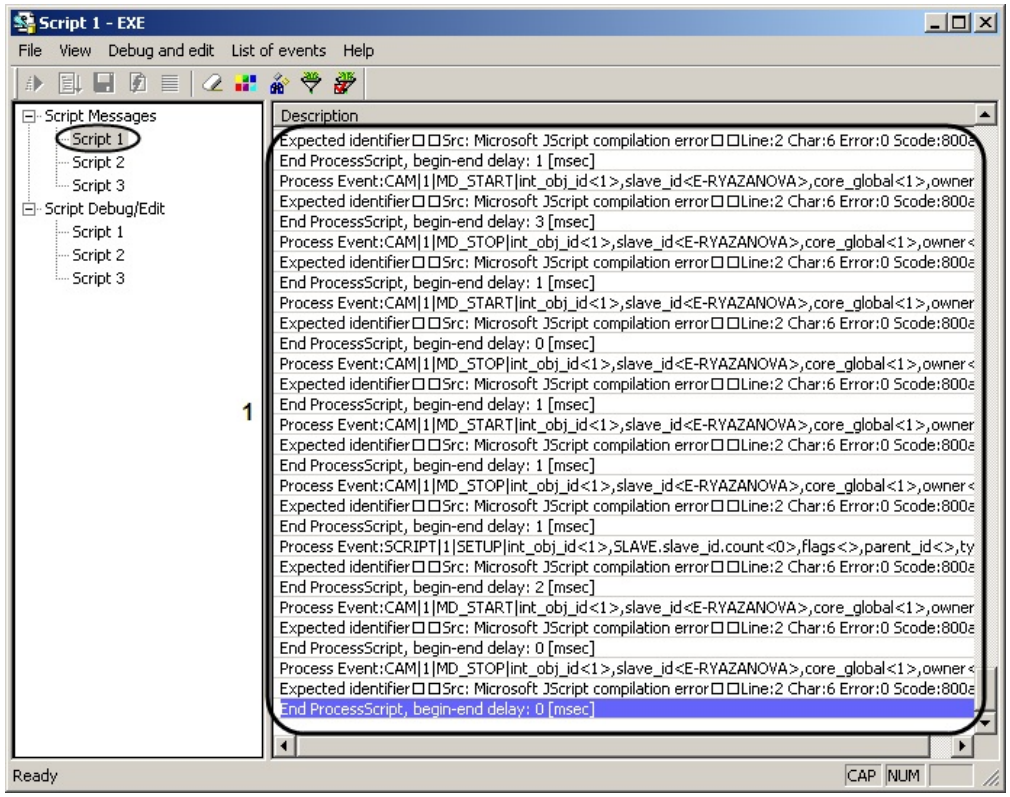


Table describes the elements in the **Script** object.

№	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	By default 200 lines are displayed in the list. The value can be changed using the DebugMaxLines registry key (see Registry keys reference guide).

Main menu

On the page:

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 describes the elements of the main menu.

№	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.	-	-	-

The elements in the File menu

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

Table describes the elements of the **File** menu.

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	-	-	-

The elements in the View menu

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

Table describes the elements of the **View** menu.

№	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

The elements of the Debug and edit menu

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

Table describes the elements of the **Debug and edit** menu.

№	Element	Element	Description	Units	Default	Value

	name	type			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	Edit test event	Item	Opens the window for editing test events	-	-	-
4	Summary information	Item	Opens the Thread Information window showing system, test and user messages related to the current script	-	-	-
5	Go to line	Item	Opens the window for entering the script line and character number to go to	-	-	-

The Message list menu elements

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

Table describes the elements of the **Message list** menu.

№	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	-	-	-

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 shows the Filter window.

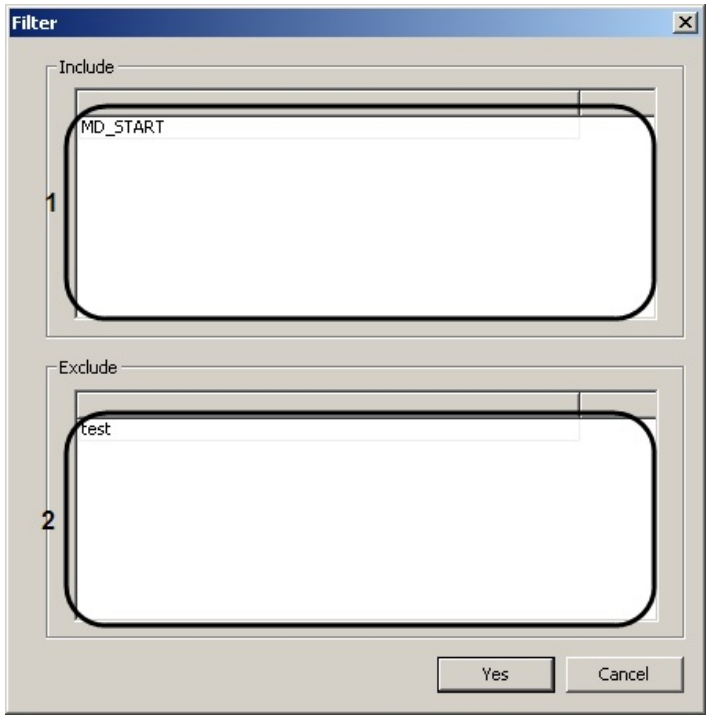


Table describes the elements in the Filter window.

№	Element name	Element type	Description	Units	Default value	Value range
1	Include	Text field	Only the lines containing the	Latin, Cyrillic and	Empty	Unlimited number

			words from this field will be displayed in the debugger window	special symbols		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

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 lis**t menu, select **Colors**.

Figure shows the Color window.

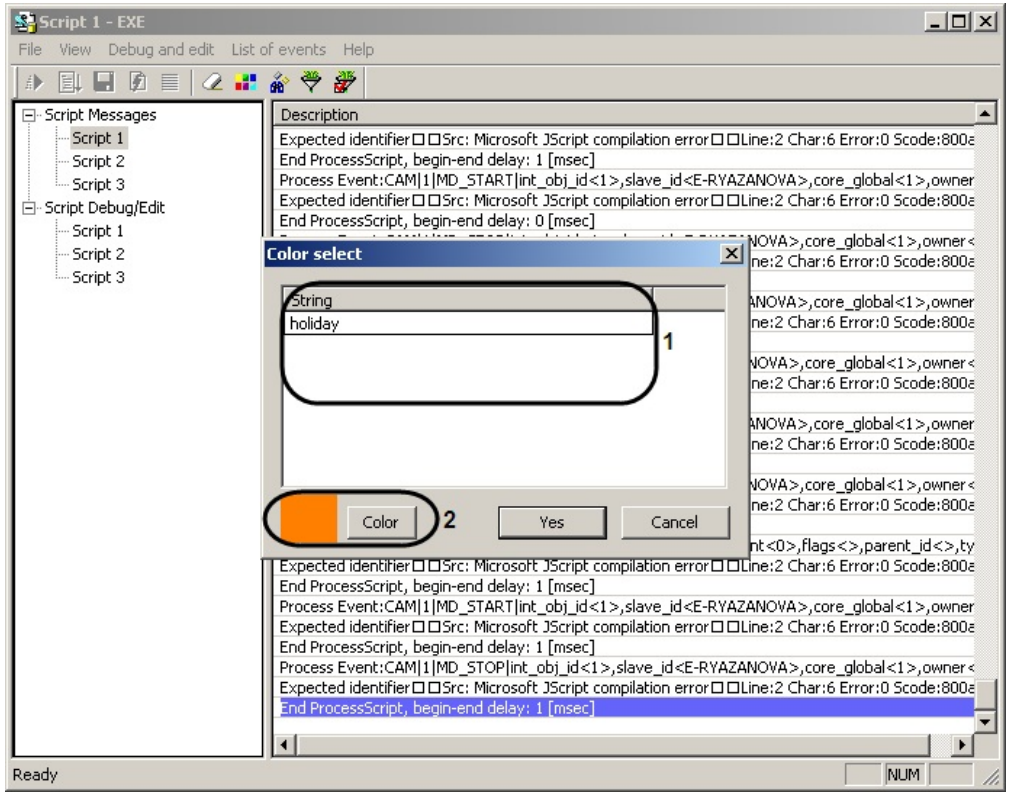


Table describes the elements in the **Color** window.

№	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

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 shows the toolbar in the script editing mode.

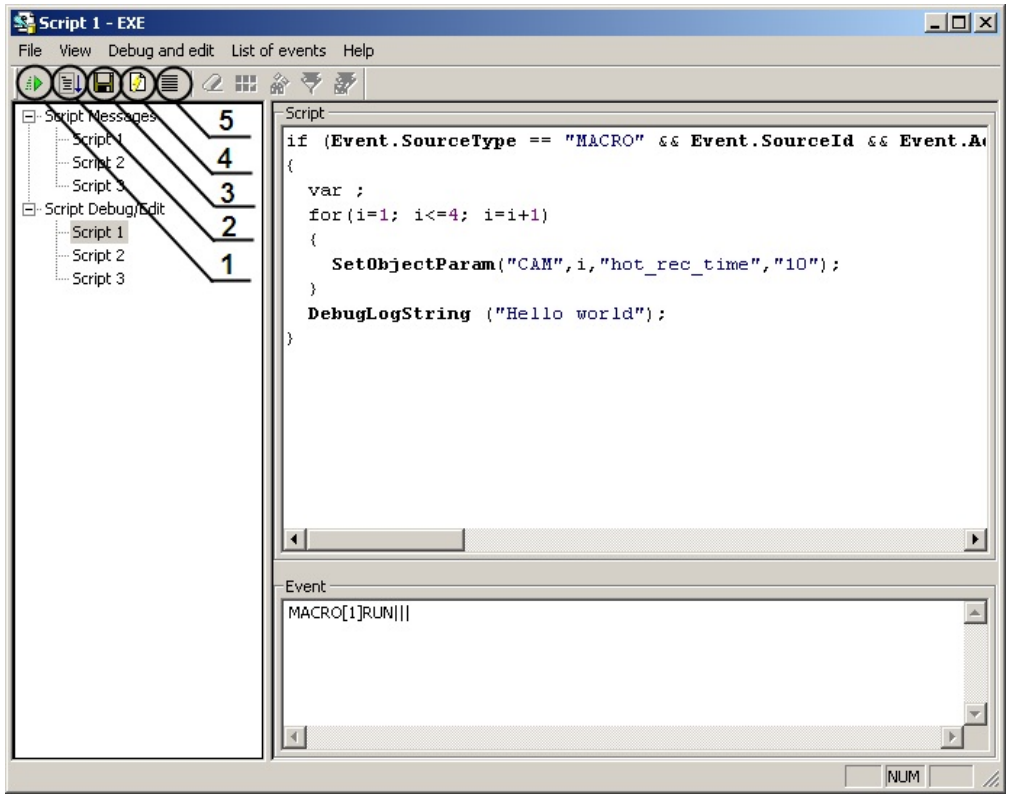


Table shows the elements of the toolbar in the script editing mode.

№	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	Button	Runs the script using the third-party debugger	-	-	-

	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 shows the toolbar in the script debugging mode.

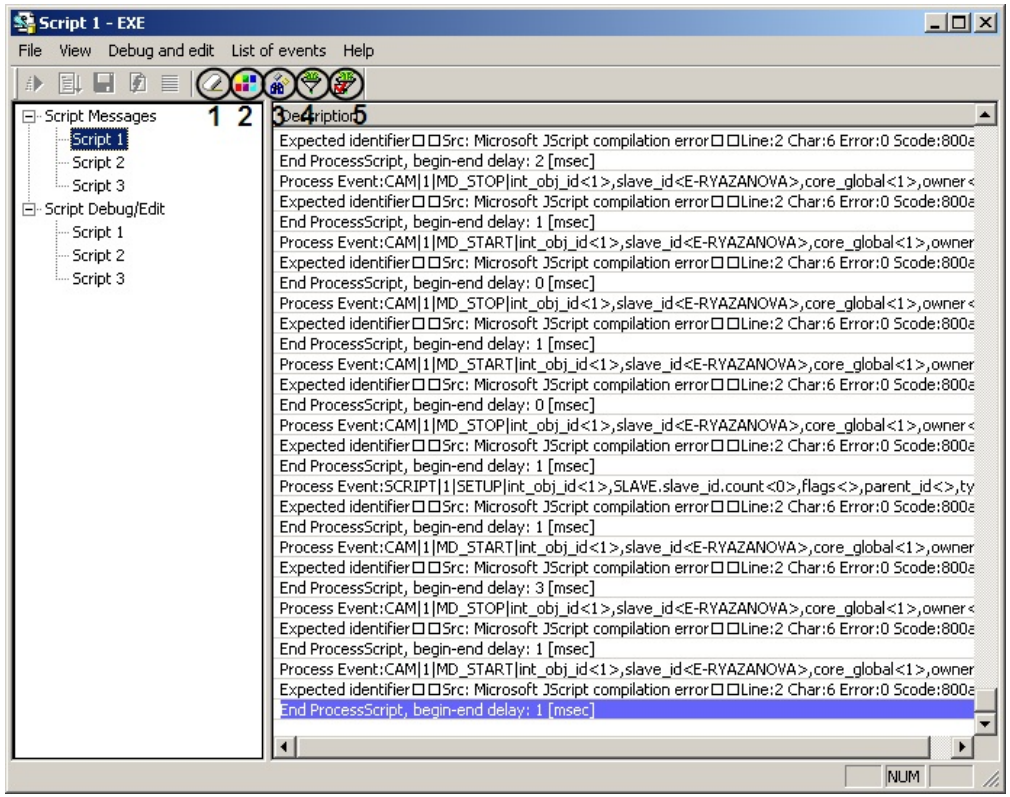


Table shows elements of the toolbar in the script debugging mode.

№	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	-	-	-

Appendix 2. Creating virtual objects with ability to set events, reactions and states

Purpose of virtual objects and their implementation in Intellect

Virtual objects represent software emulation of new *Intellect* objects and allow configuring their states, reactions and events. Virtual objects are handled using scripts, macrocommands and macroevents.

Virtual objects are created using the ddi.exe and CustomTypeEditor.exe utilities in the <*Intellect* installation folder>\Tools.

In the How to create a virtual object section you can find an example of creating 2 types of virtual objects that can be used to show the state of abandoned object detection tool on the map or to show any other user states. The states of objects are changed using macros, scripts or via IIDK.

How to create a virtual object

Here one can find out how to create the following virtual objects:

1. CUSTOM type with SLAVE (Computer) parent type.
2. CUSTOM_CHILD type with CUSTOM parent type (see item 1).

An object of the CUSTOM type has the property set:

1. Custom_param1 and custom_param2 parameters
2. Events: ALARM, INFO, ON, OFF
3. Reactions: ON, OFF
4. States: ON, OFF
5. State machine:
    1. Set ON state for ON event
    2. Set OFF state for OFF event

The CUSTOM_CHILD type of the object is created to demonstrate tree structure and has no user parameters, events, reactions or states.

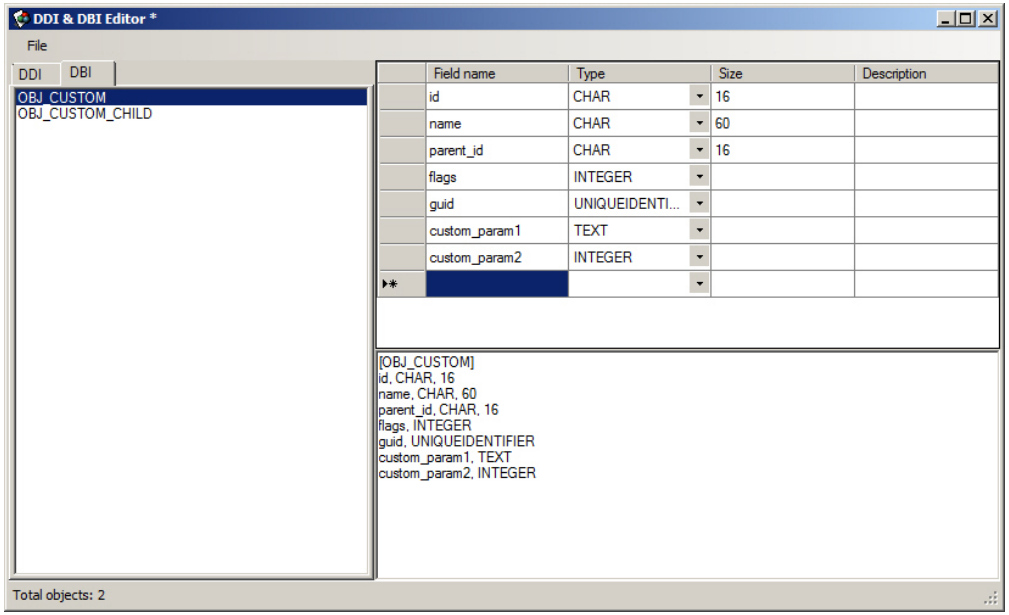
dbi file preparation

A dbi file is prepared using the ddi.exe utility. Details on how to handle it can be found in the The ddi.exe utility for editing database templates and external settings files section of Administrator's Guide.

A dbi file for objects of CUSTOM and CUSTOM_CHILD type is created as follows:

1. Run dbi.exe (see The ddi.exe utility for editing database templates and external settings files).
2. Go to the **DBI** tab.
3. Create two objects – OBJ_CUSTOM and OBJ_CUSTOM_CHILD – as shown in the figure below.

> **Important!**
>
> Object (table) names should look like OBJ_<object type>.



> **Note.**
>
> Id, name, parent_id, flags and guid parameters are mandatory for all objects. Custom_param1 and custom_param2 are user parameters.

4. Save changes using the **Save** command in the **File** menu. The saved file must have dbi extension and must be located in *Intellect* installation directory - C:\Program Files (x86)\Intellect\intellect.custom.dbi

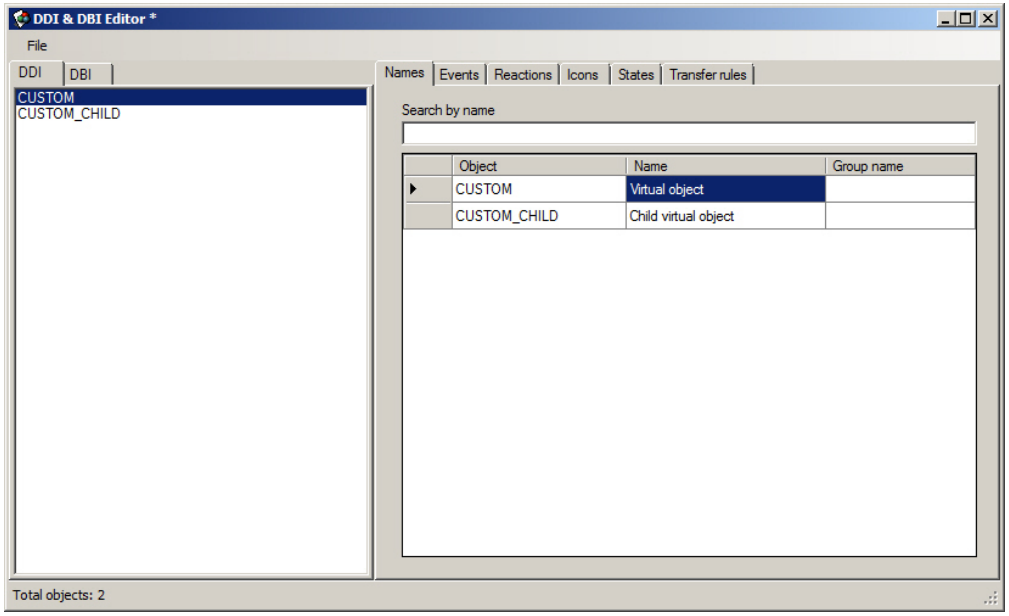
dbi file preparation is complete.

ddi file preparation

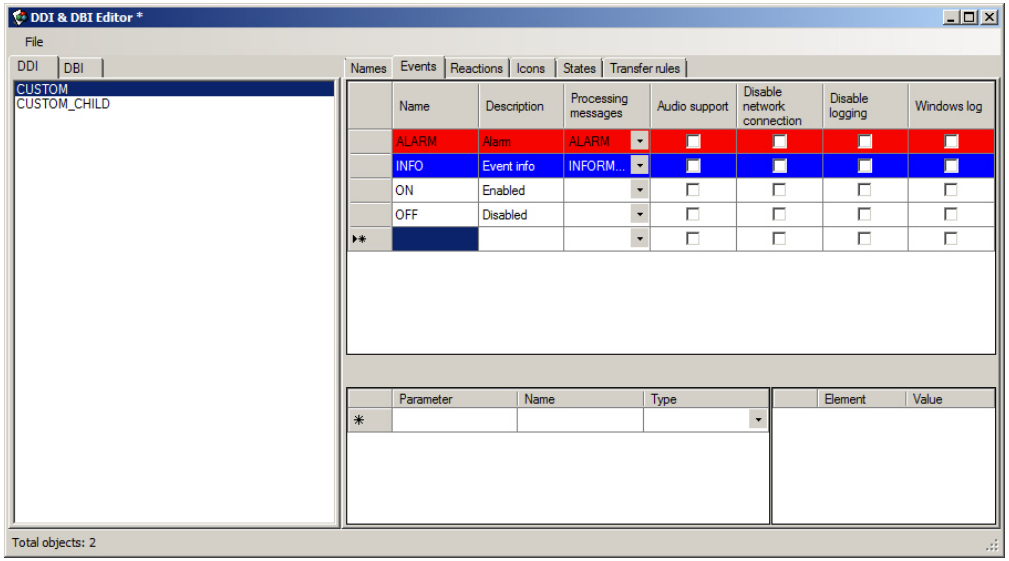
A ddi file is prepared using the ddi.exe utility. Details on how to handle it can be found in the The ddi.exe utility for editing database templates and external settings files section of Administrator's Guide.

A ddi file for CUSTOM and CUSTOM_CHILD object types is created as follows:

1. Run ddi.exe (see The ddi.exe utility for editing database templates and external settings files).
2. Create CUSTOM and CUSTOM_CHILD objects in the **DDI** tab as shown below.

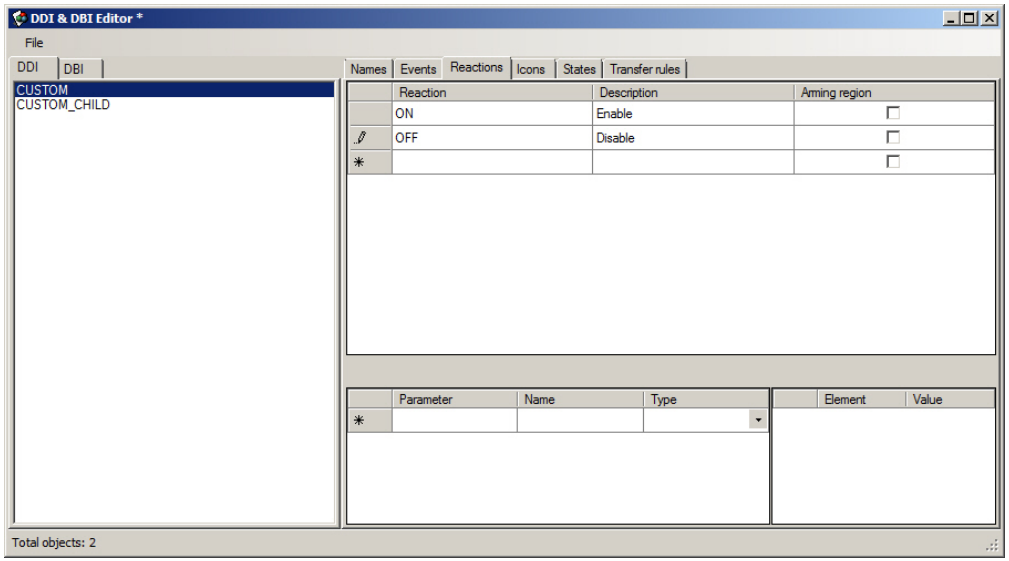


3. Go to the **Events** tab and configure events that are to be supported by the object (see the figure).

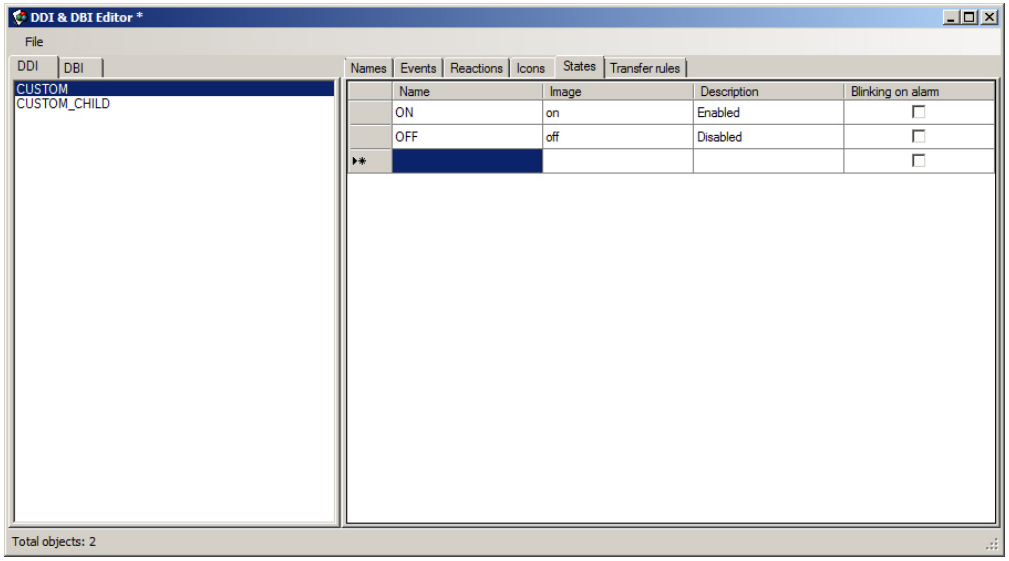


4. Go to the **Reactions** tab and configure reactions that are to be supported by the object (see the figure).

> **Note.**
>
> Reactions of virtual objects are automatically converted into events. In other words, a virtual object automatically generates an event when there is a reaction.



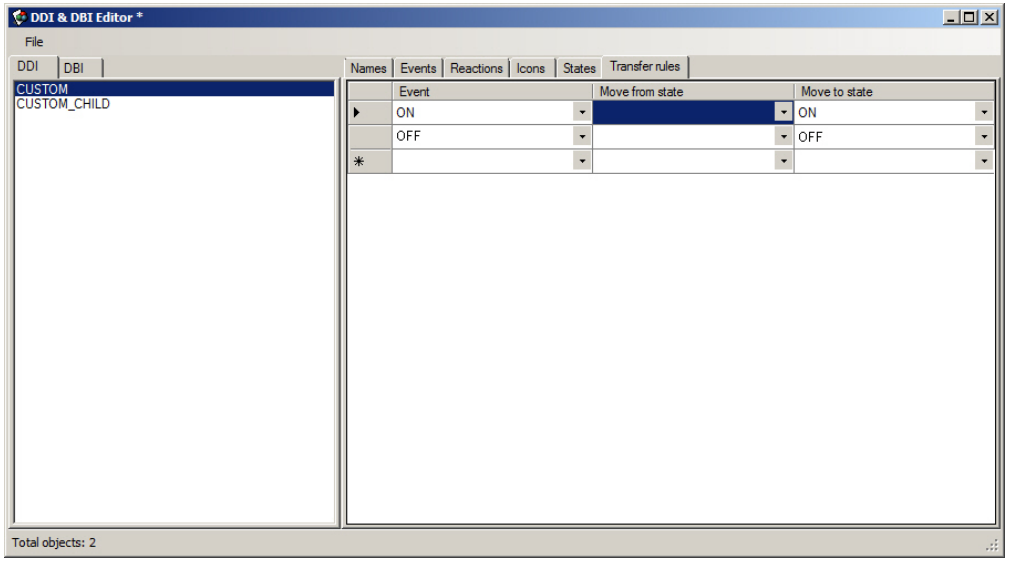
5. Go to the **States** tab and describe the states that the object can take. Here there are two states – ON and OFF.



Note.

The postfix of file name is specified in the **Image** column – the image that is stored in < *Intellect* installation directory>\Bmp. For instance, these will be custom_off.bmp and custom_on.bmp files (corresponding to ON and OFF states) for CUSTOM object. These files will be used by the map module.

6. Go to the **Transition rules** tab and set the object state change logic.



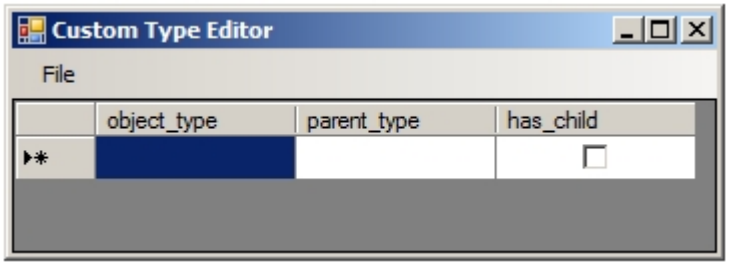
Transition rules is a simple state machine – an event is an input action and a state is a result.
An unconditional transition is used in this case: if CUSTOM||ON event is received, then there is transition to the ON state, if CUSTOM||OFF event is received, then there is transition to the OFF state.

7. To save changes use the **Save** command in the **File** menu. The saved file must have the ddi extension and be stored in the folder corresponding to the required language, e.g. C:\Program Files (x86)\Intellect\Languages\en\intellect.custom.ddi

The ddi file preparation is completed.

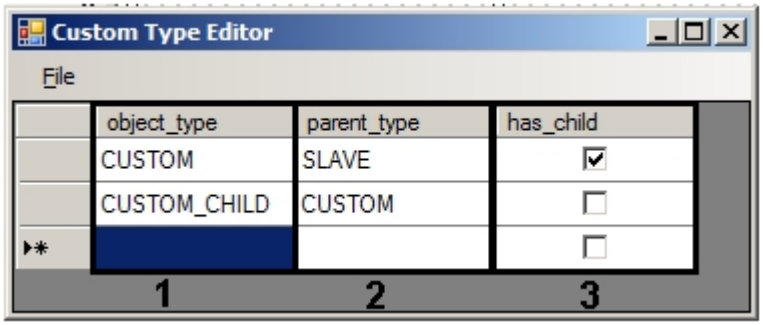
XML file preparation

An XML file is prepared using the CustomTypeEditor.exe utility that can be found in <*Intellect* installation directory>\Tools. The utility overview is shown below.



An XML file for a virtual object is created as follows:

1. Specify the name of object type in the **object_type** field (1).



2. Specify the name of parent type in the **parent_type** field (2).
3. If the object type has child types, then set the **has_child** checkbox checked (3).
4. Repeat steps 1-3 for all object types.
5. Save a file with any name in the *Intellect* installation directory using the **File** – **Save** command.

The XML file is now created. The file contents look like this:

```
<?xml version="1.0" standalone="yes"?>
<objects>
<object>
<object_type>CUSTOM</object_type>
<parent_type>SLAVE</parent_type>
<has_child>1</has_child>
</object>
<object>
<object_type>CUSTOM_CHILD</object_type>
<parent_type>CUSTOM</parent_type>
</object>
</objects>
```

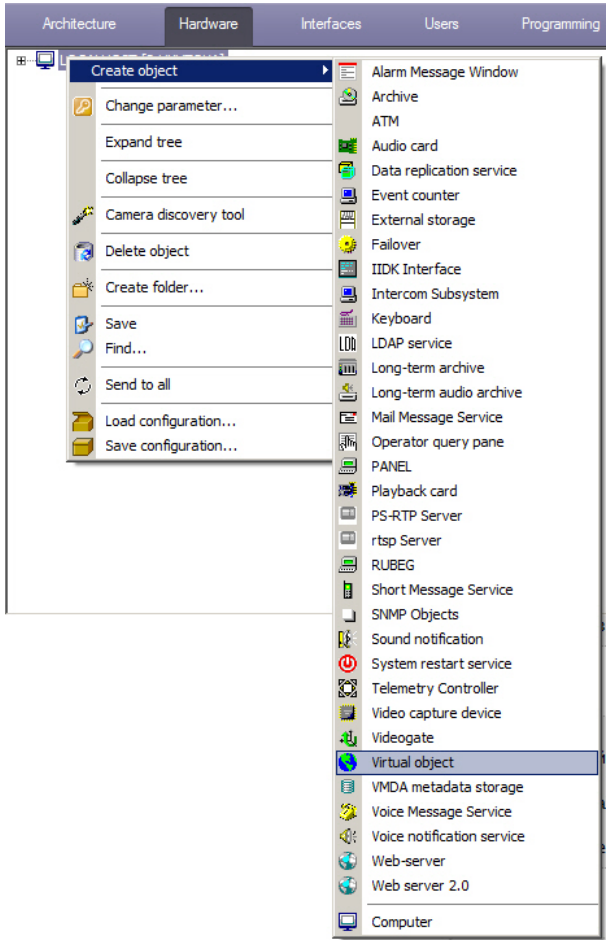
Modify it manually if required.

Creating and using a virtual object in Intellect

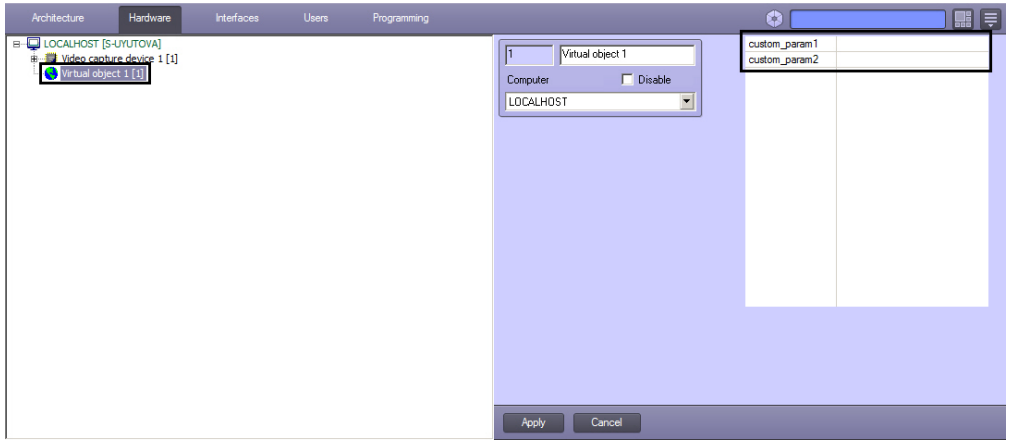
On the page:
• Showing on the map

- Using in macros
- Sample program in Jscript to change virtual object state

When dbi, ddi and XML files are ready objects of a new type along with standard objects can be created in *Intellect* hardware tree.

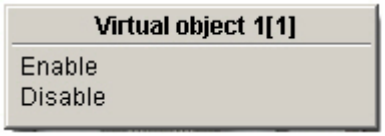


User parameters are displayed on the settings panel of the created virtual object – custom_param1 and custom_param2 in this example. Their values can be set in the table.



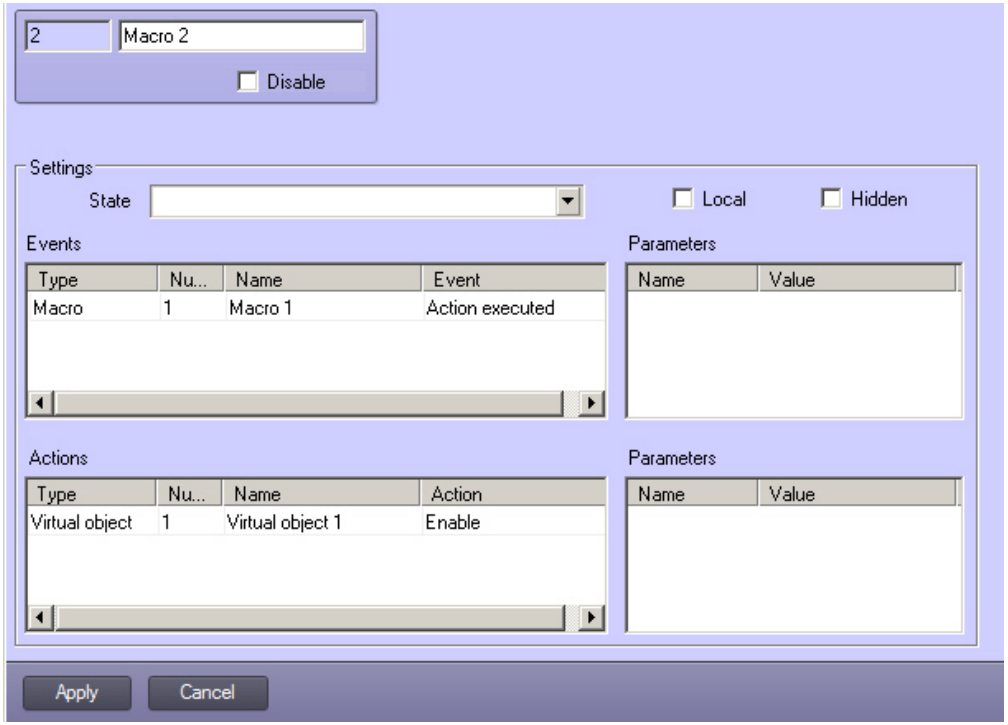
Showing on the map

When an object is created in the hardware tree it can be placed on the map and set reactions can be executed in the object context menu (see Configuring the interactive map for object state indication and controlling the objects).



Using in macros

When a virtual object is created in the hardware tree it can be used in macros.



Note.

Reactions of virtual objects are automatically converted into events. Thus, in the sample when **ON** reaction is executed the object state changes thanks to set state transition rules (see ddi file preparation) and the icon corresponding to the state will be shown on the map.

Sample program in Jscript to change virtual object state

Problem. Using macro 1 change a state of virtual object 1 to ON and show the icon corresponding to this state on the map.

Solution. As state transition rules are set, when ON event is sent from the virtual object the state will be automatically changed to ON and the icon specified in ddi file (see ddi file preparation) to this state will be shown on the map. A script for sending ON event looks like this:

```
if (Event.SourceType == "MACRO" && Event.SourceId == "1" && Event.Action == "RUN")

{
```

```
    var msgevent = CreateMsg();
    msgevent.SourceType = "CUSTOM";
    msgevent.SourceId = "1";
    msgevent.Action = "ON";
    NotifyEvent(msgevent);
}
```