Intellect Software Integration Guide (HTTP API, IIDK, ActiveX)

1. Intellect Software Integration Guide. Introduction 5
2. Hardware and Software Module Integration 5
2.1 Integrating hardware and software modules with Intellect 5
2.1.1 General information on hardware and software modules integrating 5
2.1.2 Editing the DBI file 6
2.1.2.1 Adding Objects to Intellect.dbi 6
2.1.2.2 Using the ddi.exe Tool to Work with DBI files 10
2.1.3 Editing the DDI file 12
2.1.3.1 Adding Object Information to intellect.ddi 13
2.1.3.2 Using the ddi.exe Tool to Work with DDI files 15
2.1.4 Additional Functionality of the ddi.exe Utility 19
2.1.5 Creating MDL files 20
2.1.5.1 MDL File Creation Wizard 30
2.1.6 Creating RUN files 31
2.1.7 Creating and Configuring Integrated Objects (Modules) in Intellect 32
2.2 Intellect Integration Developer Kit (IIDK) 33
2.2.1 General Information on IIDK 34
2.2.1.1 Purpose of the IIDK 34
2.2.1.2 Developer Requirements 34
2.2.1.3 IIDK Components 34
2.2.2 Connecting to Intellect 35
2.2.2.1 Connection Parameters 35
2.2.2.2 IIDK Interface Object 35
2.2.2.3 Configuring passing events throung IIDK Interface object 36
2.2.2.4 Features of ATMs integration. ATM object 37
2.2.3 IIDK Functions 37
2.2.3.1 Connect 37
2.2.3.2 SendMsg 39
2.2.3.3 Disconnect 40
2.2.3.4 Other functions 40
2.2.4 Sent Message Syntax 43
2.2.4.1 Message Syntax 43
2.2.4.2 Message Syntax (port 900) 44
2.2.4.3 Using the Event and React classes 45
2.2.5 Examples of Managing System Objects 45
2.2.5.1 Adding, Updating, and Deleting System Objects 45
2.2.5.2 Working with the System in the Multiuser Mode 46
2.2.5.3 Determining Computers Where Intellect was Unloaded (via Port 1030) 46
2.2.5.4 Redirecting Video Cameras to the Monitor 47

2.2.5.5 Obtaining Object Parameters (via Port 1030) GET_CONFIG . . . 47
2.2.5.6 Obtaining Information on Object States GET_STATE and GET_LIST . . . 48
2.2.5.7 Showing Information Messages. SET_STATE . . . 48
2.2.5.8 Live and archived video . . . 48
2.2.5.9 Telemetry control via IIDK . . . 49
2.2.5.10 Map layer operations . . . 49
3. CamMonitor.ocx ActiveX Control . . . 49
3.1 General description of CamMonitor.ocx component of ActiveX . . . 50
3.2 How to install CamMonitor.ocx . . . 50
3.3 CamMonitor.ocx parameters . . . 51
3.4 CamMonitor.ocx methods . . . 54
3.5 CamMonitor.ocx events . . . 56
4. Intellect HTTP API . . . 57
4.1 General information on HTTP API . . . 57
4.2 Getting events . . . 58
4.2.1 Getting events of video subsystem in blocks . . . 59
4.3 Maps . . . 61
4.3.1 Getting the list of maps . . . 61
4.3.2 Information on one map . . . 62
4.3.3 The list of layers for specific map . . . 62
4.3.4 Information on a specific layer . . . 63
4.3.5 Layer background . . . 64
4.3.6 The list of point on the layer . . . 64
4.3.7 Information on a specific point on the layer . . . 66
4.4 Object classes . . . 66
4.4.1 The list of object classes on the server . . . 66
4.4.2 Specific object class . . . 67
4.4.3 The list of states for a specific object class . . . 67
4.4.4 Information on a specific state . . . 68
4.4.5 Getting the icon for a specific state . . . 68
4.4.6 The list of events for a specific object class . . . 68
4.5 Objects . . . 70
4.5.1 Getting list of objects . . . 70
4.5.2 Information on a specific object . . . 72
4.5.3 State of a specific object . . . 72
4.5.4 The list of available actions with the object in a specific state . . . 74
4.6 Product version . . . 74
4.7 Sending commands to server . . . 74
4.8 Macros . . . 75

4.9 Video . . . 76
4.9.1 Configuration request . . . 76
4.9.2 Video query . . . 78
4.9.3 Format of main stream . . . 78
4.9.3.1 Managing records . . . 80
4.9.3.2 Arming and disarming the camera . . . 81
4.9.3.3 PTZ control . . . 81
4.9.4 Thumbnails request . . . 82
4.10 Using the archive . . . 83
4.11 Notification . . . 89
4.12 Sound . . . 90
4.12.1 Getting live sound . . . 90
4.12.2 Playing sound from archive . . . 92
4.12.3 Sending live sound . . . 92
4.13 Commands used for ECHD integration . . . 93
4.13.1 Archive downloading . . . 93
4.13.2 Archive export . . . 93
4.13.3 List of cameras and their parameters . . . 95
4.13.4 Ranges of available archive recordings . . . 96
4.13.5 Video surveillance device features management . . . 97
4.13.6 Working with video streams . . . 102
4.14 Sending reactions and events to Intellect using HTTP request . . . 103
4.15 Sending HTTP API commands using curl tool . . . 103
5. Intellect software Integration Guide. Postscript . . . 103
6. APPENDIX 1. DDI file structure . . . 104
7. APPENDIX 2. NissObjectDLLExt and CoreInterface class declarations . . . 106

Intellect Software Integration Guide. Introduction

This document provides the information needed for Intellect software interaction with external systems. Intellect software has the following interfaces for this:

1. IIDK Interface: intended for integration of functional modules that perform the following tasks:
    a. Adding new security hardware to the system.
    b. Implementing new service functions (security hardware management).

    The module integration steps are explained using a demonstration module, DEMO, as an example (its source code may be found in an appendix to the documentation). DEMO module can also be downloaded in section Intellect Software Integration Guide (HTTP API, IIDK, ActiveX) of the online documentation.
2. CamMonitor.ocx ActiveX Control: the component is similar in every way to the Video monitor interface object. It allows you to manage cameras, view the archive, etc.
3. HTTP API: the program interface allows to send commands and receive data from Intellect software by HTTP requests.

Hardware and Software Module Integration

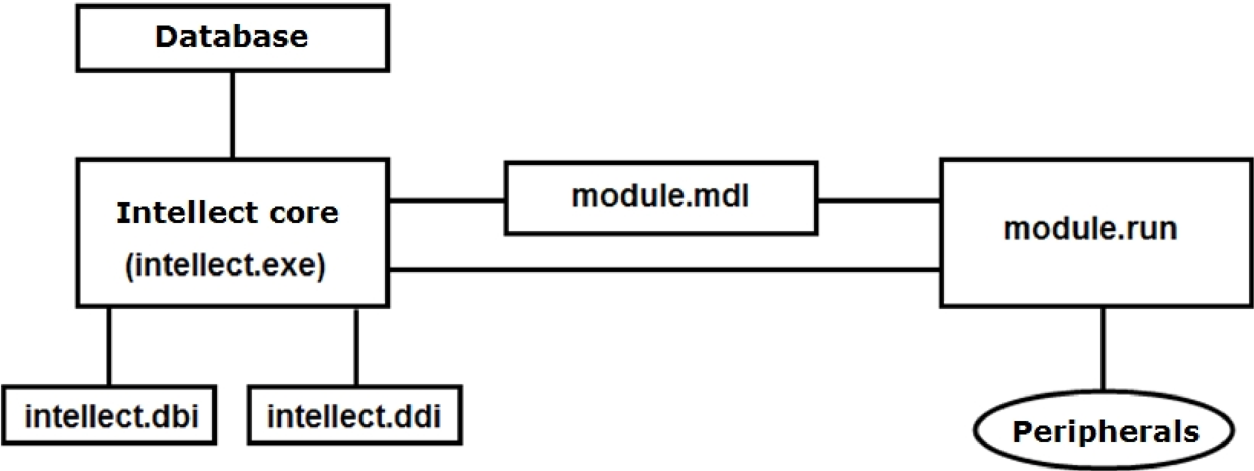
Integrating hardware and software modules with Intellect

General information on hardware and software modules integrating

To integrate a hardware and software (functional) module into Intellect, perform the following steps:

1. Edit the DBI file.
2. Edit the DDI file.
3. Prepare a module.mdl file, where “module” is the name of the module to be integrated (this file is a transformed DLL file).
4. Prepare an executable file, module.run, where “module” is the name of the module to be integrated (this file is a transformed EXE file).
5. Copy module.mdl and module.run to the *Intellect\Modules* folder.

Figure below shows a diagram of interaction between a functional module and the system core.



The DBI and DDI files contain information on the integrated functional modules (objects); this information is needed for the operation of the system core. The DBI file describes the structure of Intellect's configuration database. The DDI file describes the objects and their parameters. When an object is integrated, the name of the object, its parameters, and the related system events and reactions are added to these files.

The MDL file is used for working with one type of objects: it allows you to create, delete, and modify object parameters (during setup or operation), save them in the database, and perform several special operations. The MDL file also ensures that the parameters of created/modified objects are sent to the executable file (the RUN file), and contains the configurations of the object setup panels.

The executable RUN file interacts with devices, passes event information to the core, and enables device management.

In this document, we describe the steps for module integration by using the *DEMO* module, which emulates working with virtual hardware. This module includes devices with unique addresses for accessing and polling these devices. Thus the system includes a configuration consisting of 2 main objects: a parent object, **DEMO,** with the **COM port** parameter, and a child object, **DEMO_DEVICE**, with the **Address** parameter. The system allows you to perform a number of actions with devices and to pass all their events to the system core.

Editing the DBI file

The intellect.dbi file contains the master list of the tables and fields of the database. We recommend that you create your own database template in a separate file and name it intellect.xxx.dbi, where xxx is a unique sequence in the filename. By using a separate file, you avoid double inclusion of the tables and fields after an update to the Intellect software package. On startup of the software package, the DBI files are merged.

Adding Objects to Intellect.dbi

Objects are added to intellect.dbi as follows:

1. Go to *Intellect's* root folder and open the intellect.dbi file with a text editor.
2. Add the objects to intellect.dbi. For each object, you must supply its name (used for identification) in brackets and then declare its fields. Below is the field declaration syntax:
   <Field name>, <Type> [, <Size>]

   > **Note.**
   > The **Size** may be set for fields of the CHAR type only.

The table below shows the fields mandatory for all objects in *Intellect.*

Field	Description
id	Unique object ID
name	Object name
parent_id	Parent object ID
flags	Parameter for internal system use

Attention!
The flags field may not be used by external applications.

The following table describes the allowed data types.

Data type	**Description**
BIT	Used for creating a flag field that takes a logical value, Yes or No.
CHAR	Used for fields that contain short character sequences.
DATETIME	Used for fields that contain dates and times. The date format is YYYY-MM-DD and the time format is HH:MM:SS.XXX.
DOUBLE	Used for fields that contain floating-point numbers.
INTEGER	Used for fields that contain integer numbers.
TEXT	Used for fields that contain text strings.

Beside the mandatory fields, the objects of the DEMO module contain the following fields:

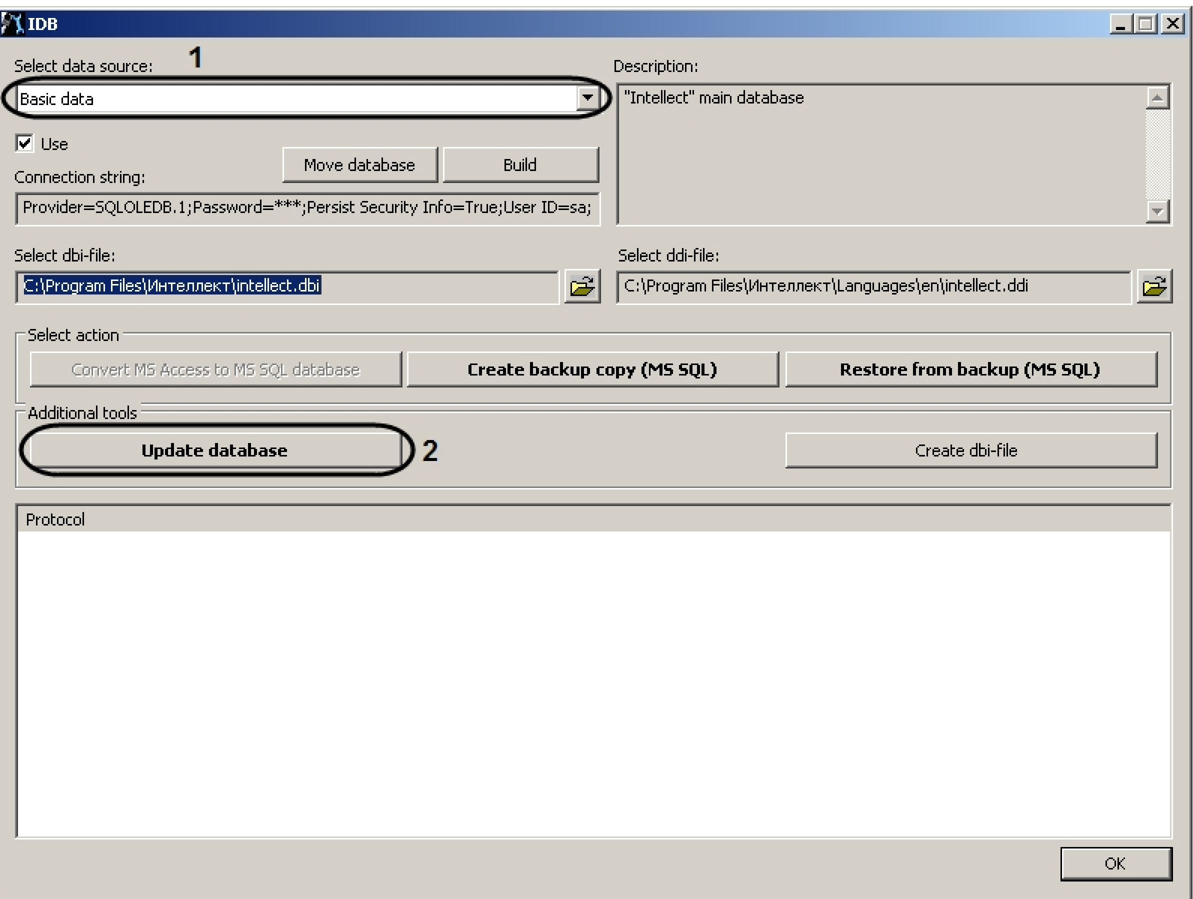
a. **port** – COM port address;
b. **address** – device address.

Figure below shows sample object additions and field declarations in intellect.dbi.

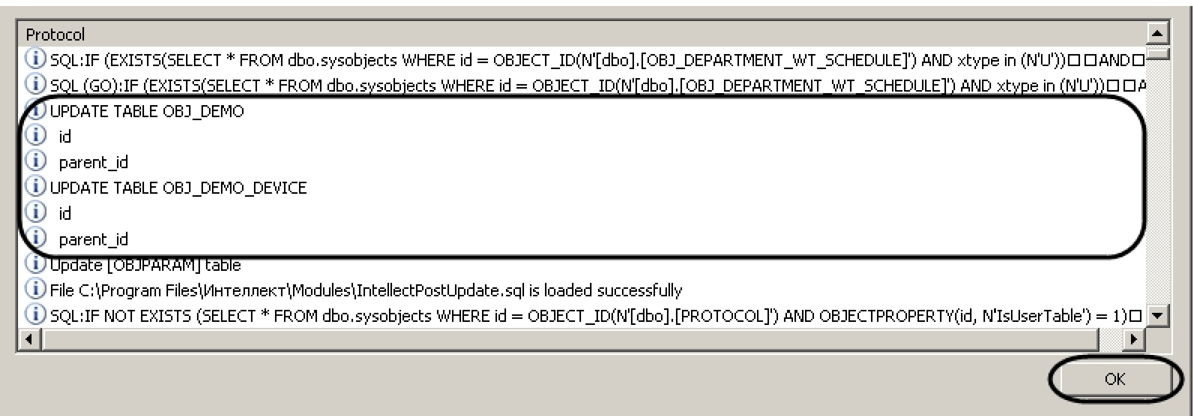
```
[OBJ_DEMO]
id, CHAR, 16
name, CHAR, 60
parent_id, CHAR, 16
flags, INTEGER
port, CHAR, 5

[OBJ_DEMO_DEVICE]
id, CHAR, 16
name, CHAR, 60
parent_id, CHAR, 16
flags, INTEGER
address, INTEGER
```

3. Save the changes to the intellect.dbi file.
4. Go to Intellect's root folder and run the idb.exe utility.



5. In the **Select data source** list, select **Basic data** (1).
6. Click the **Update database** button (2).
   The system will start updating the database structure. The progress will be shown in the **Protocol** window of idb.exe.



7. Click **OK** to close idb.exe.

As a result of the database structure update, tables are created in *Intellect's* configuration database.

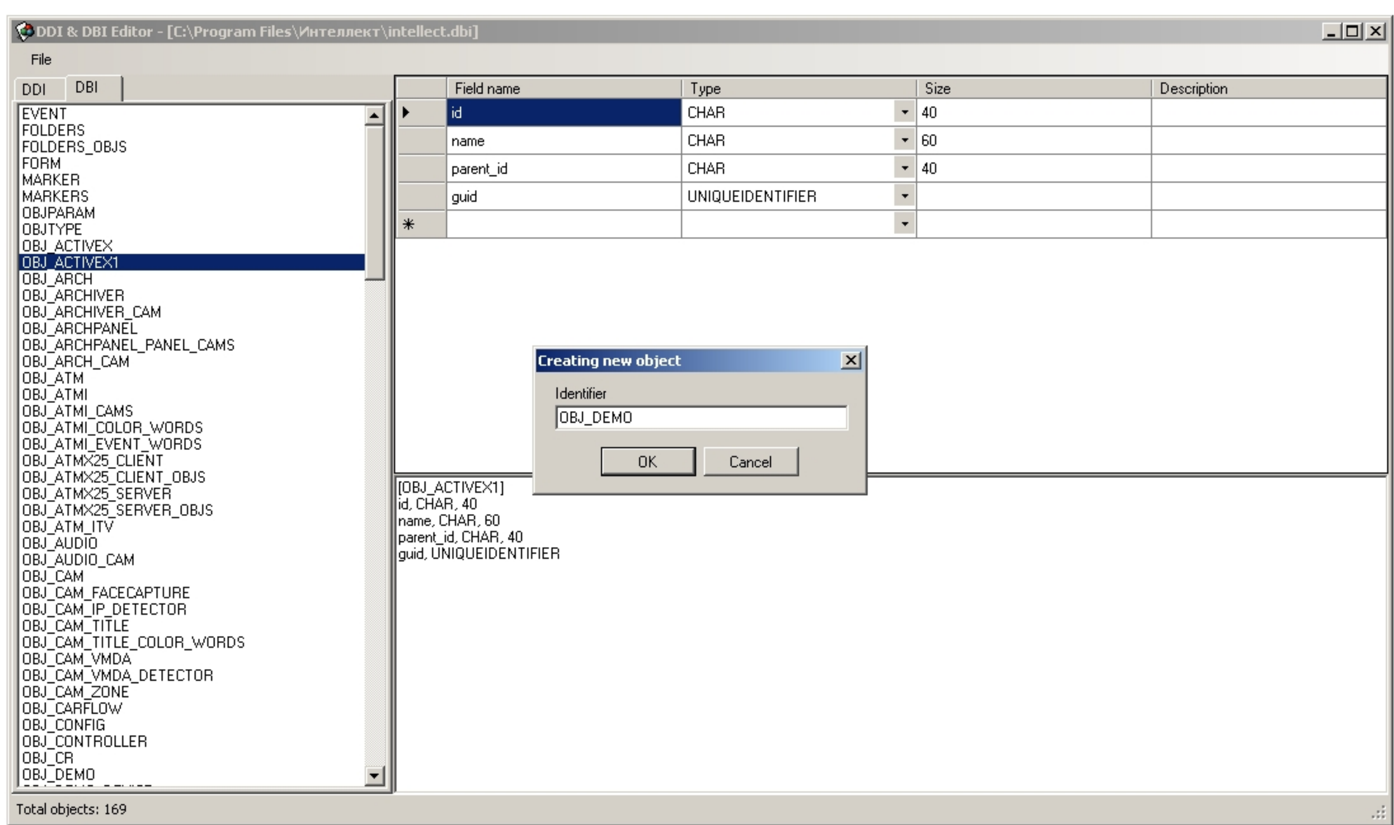
Using the ddi.exe Tool to Work with DBI files

To add an object to the DBI file by using the *ddi.exe* utility, do the following:

1. Go to the *Intellect\Tools* folder and run *ddi.exe*.
2. In the program window, select the **DBI** tab.
3. In the **File** menu, select **Open**. The **Open** dialog box appears.
4. Go to Intellect's root folder and select the intellect.dbi file. The *ddi.exe* window shows a list of objects.
5. To add the new object, in the list's context menu, select **Add**.

Note:
You may add a new object by pressing the **Insert** key as well.

6. A dialog box opens. In the **ID** field, enter an object name (used for identification) and click **OK.**

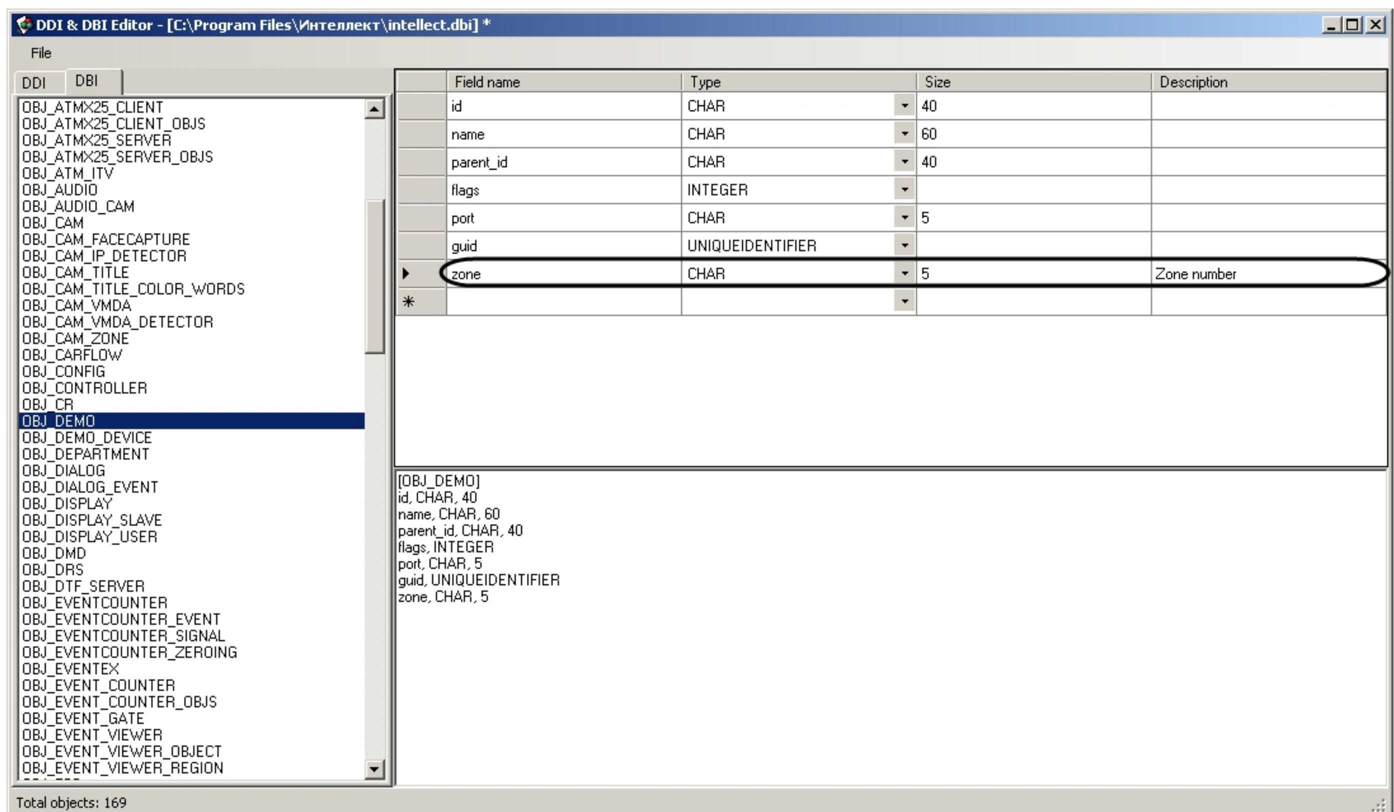


Note:
The mandatory fields are automatically added to the created object (see Section Adding Objects to Intellect.dbi).

The object has now been added to the DBI file.

To add a field:

1. In the left part of the *ddi.exe* window, select an object.
2. Add a description of the new field (a string) to the table.



3. To save the changes, in the **File** menu, select **Save.**

The new field is now added.

Attention!
After making changes to the DBI file, you must update the database structure by using idb.exe (see Adding Objects to Intellect.dbi).

Editing the DDI file

The DDI file is an XML file that contains the following object information:

1. Reactions (that is, actions that the objects may perform).
2. Events that the objects may generate.
3. States of the objects.
4. Event-driven rules for state transition.
5. The names of the BMP files that are used for visualizing the objects on the *Map*.

The intellect.ddi file contains the properties of Intellect's main objects. For your own objects, we recommend creating a separate file, intellect.xxx.ddi, where xxx is a unique part of the filename. By using a separate file, you avoid double inclusion of the properties after an update of the Intellect software package. On startup of the software package, the DDI files are

merged.

If an object is duplicated in several ddi-files, then at Intellect software startup the properties of the object from the last file are applied in accordance with the sorting of files by name. For example, if an object is described in files intellect.xxx.ddi, intellect.xxx1.ddi and intellect.xxx2.ddi, the properties from the intellect.xxx2.ddi will be applied.

Adding Object Information to intellect.ddi

This section shows how to add information on the **DEMO** object to intellect.ddi by using a text editor.

To add information on the **DEMO** object, do the following:

1. Go to *Intellect\Languages*\en folder and open the intellect.dbi file with a text editor.
2. Into the **<DataSetDDI>** section, add a child element, **<Objects>**, which contains an object description.

```xml
<Objects>
  <ObjectName>DEMO</ObjectName>
  <VisibleName>Demo object</VisibleName>
  <GroupName></GroupName>

  <Events>
    <EventName>LOST</EventName>
    <EventDescription>Connection lost</EventDescription>
    <IsSoundEnabled>false</IsSoundEnabled>
    <IsNetworkDisabled>false</IsNetworkDisabled>
    <IsProtocolDisabled>false</IsProtocolDisabled>
    <IsWindowsLogEnabled>false</IsWindowsLogEnabled>
  </Events>
  <Events>
    <EventName>RESTORE</EventName>
    <EventDescription>Connection restored</EventDescription>
    <IsSoundEnabled>false</IsSoundEnabled>
    <IsNetworkDisabled>false</IsNetworkDisabled>
    <IsProtocolDisabled>false</IsProtocolDisabled>
    <IsWindowsLogEnabled>false</IsWindowsLogEnabled>
  </Events>
  <Icons>
    <FileName>demo</FileName>
    <IconName>demo</IconName>
  </Icons>
  <States>
    <StateName>DETACHED</StateName>
    <ImgName>detached</ImgName>
    <StateDescription>Armed</StateDescription>
    <IsStateFlashing>false</IsStateFlashing>
  </States>
  <States>
    <StateName>NORMAL</StateName>
    <ImgName>normal</ImgName>
    <StateDescription>Disarmed</StateDescription>
    <IsStateFlashing>false</IsStateFlashing>
  </States>
  <Rules>
    <EventName>RESTORE</EventName>
    <FromStateName>DETACHED</FromStateName>
    <ToStateName>NORMAL</ToStateName>
  </Rules>
  <Rules>
    <EventName>LOST</EventName>
    <FromStateName>NORMAL</FromStateName>
    <ToStateName>DETACHED</ToStateName>
  </Rules>
</Objects>
```

Note.
For the **DEMO** object, the <**Reacts**> sections is missing, because this object does not perform any actions.

Note.
The DDI file elements are described in detail in APPENDIX 1. DDI file structure.

3. Save the changes to the intellect.ddi file.

The information on the **DEMO** object has now been added to intellect.ddi.

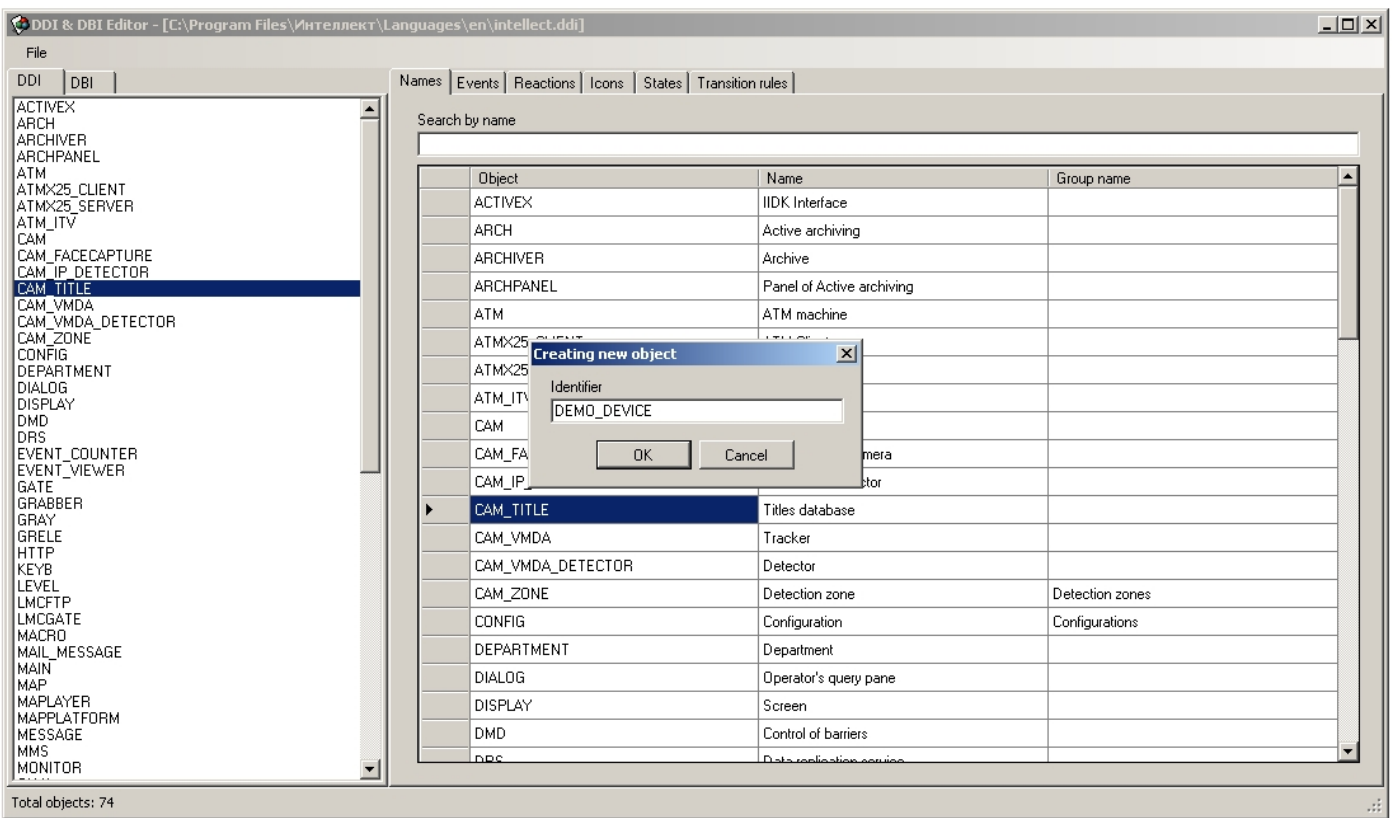
Attention!
After making changes to the DDI file, you must update the database structure by using idb.exe (see Steps 4–7 in Section Adding Objects to Intellect.dbi).

Using the ddi.exe Tool to Work with DDI files

This section shows how to add information on the **DEMO_SERVICE** object to intellect.ddi by using the *ddi.exe* tool.

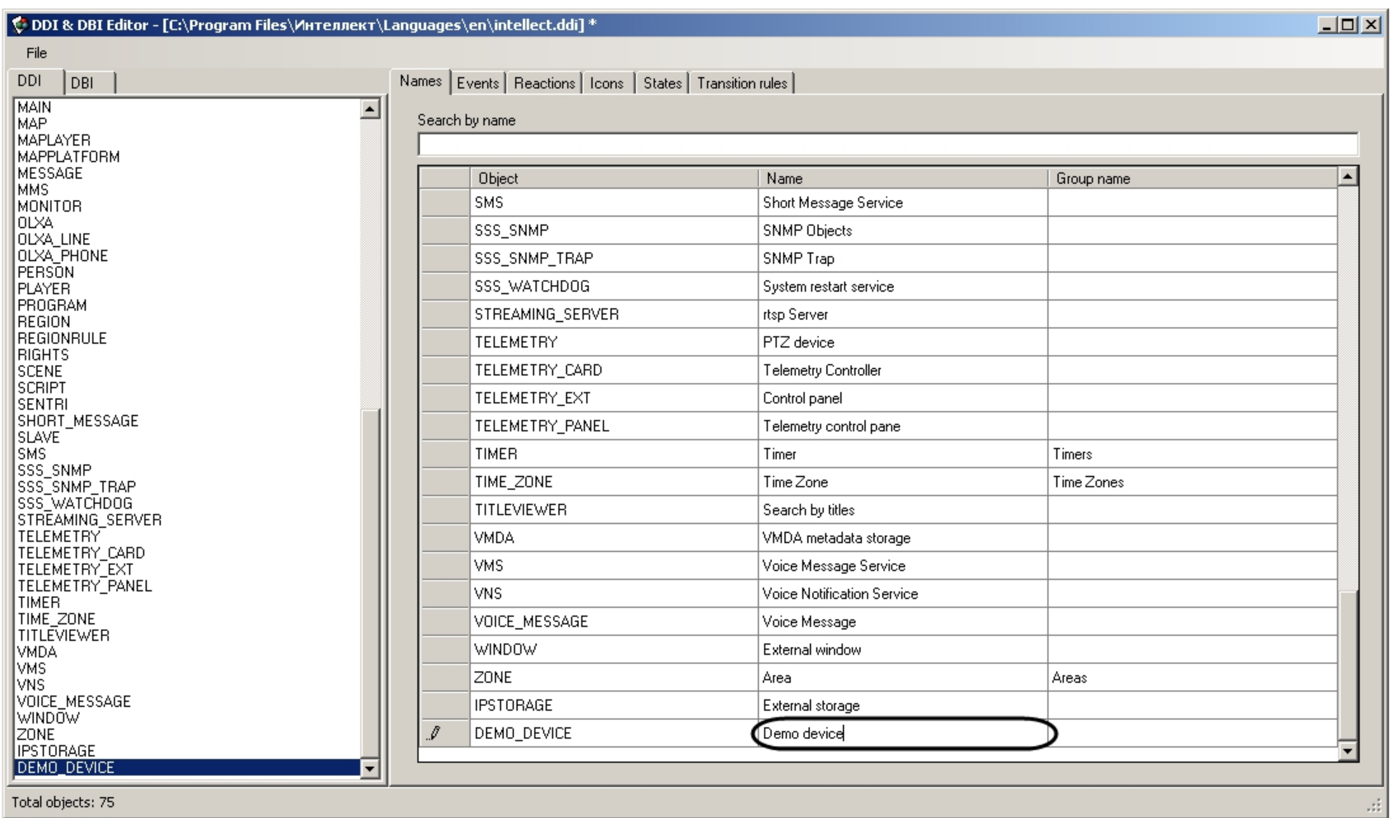
To add information on the **DEMO_DEVICE** object, do the following:

1. Go to the *Intellect\Tools* folder and run *ddi.exe*.
2. In the program window, select the **DDI** tab.
3. In the **File** menu, select **Open**. The **Open** dialog box appears.
4. Go to the *Intellect\Languages*\en folder and select intellect.ddi. The window of *ddi.exe* shows a list of objects.
5. Add the object by selecting **Add** in the list's context menu or by pressing the **Insert** key.
6. A dialog box opens. In the **ID** field, enter an object name (used for identification) and click **OK**.

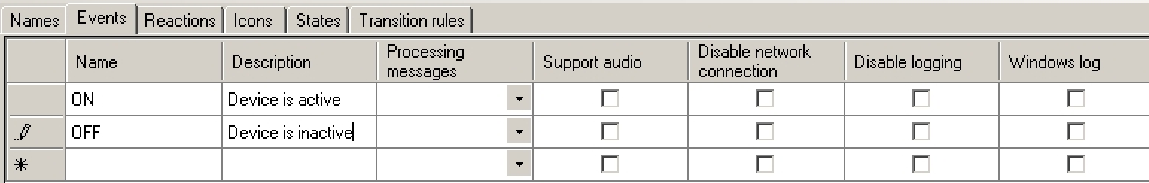


The DEMO_DEVICE object is now shown in the list of objects.

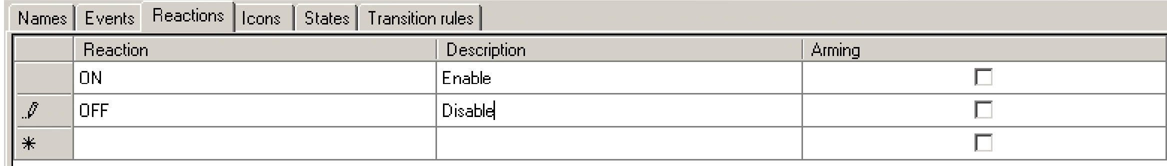
7. In the **Names** tab, enter an object name.



8. In the relevant tabs, add information on the DEMO_DEVICE object.
   a. In the **Events** tab, add the **ON** and **OFF** events.



   b. In the **Reactions** tab, add the **ON** and **OFF** actions.



   c. In the **Icons** tab, enter a BMP file name (the part that serves as an image ID). Image IDs allow you to use multiple BMP files to show objects of the same type on the **M ap.**

Names | Events | Reactions | Icons | States | Transition rules

	File name	Name
✎	demo_device	DEMO module
*		

d. In the **States** tab, add the **ON** and **OFF** states. To show an object state on the **Map,** enter a BMP file name (the part that serves as an ID of the state).

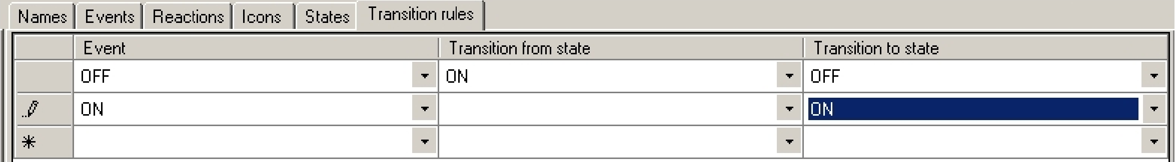
Names | Events | Reactions | Icons | States | Transition rules

	Name	Image	Description	Flicker when alarm
	ON	on	Enabled	☐
✎	OFF	off	Disabled	☐
*				☐

> **Note.**
> The names of the BMP files in the Intellect\Bmp folder must have the following format:
> <Image ID>_<State ID>
> If an image ID is not set, the BMP file name must be the following:
> <Object ID>_<State ID>

> **Note.**
> The Map may show objects using lines (that is, without using BMP files). In this case, when an object changes its state, the line color changes. For a state, the color (RGB) is set as follows:
> <State>$R:G:B

e. In the **Transition Rules** tab, set a rule for transitioning from one state to another after a certain event.



Names | Events | Reactions | Icons | States | Transition rules

	Event	Transition from state	Transition to state
	OFF	ON	OFF
✎	ON		ON
*			

> **Note.**
> If the **Transition from state** field is left blank, the rule will apply to all starting states.

9. To save changes, in the **File** menu, select **Save.**

The information on the DEMO_DEVICE object is now added.

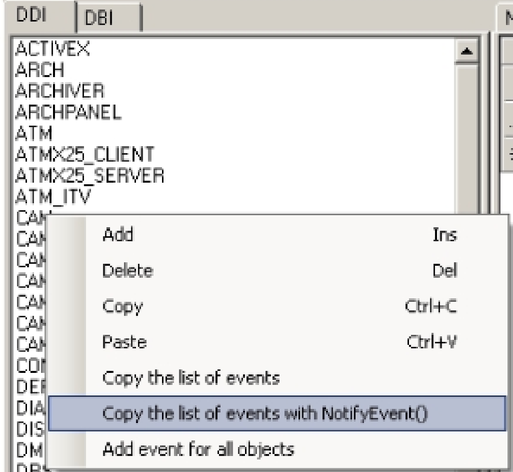
> **Note:**
> The fields of the ddi.exe tables are described in detail in APPENDIX 1. DDI file structure.

> **Attention!**
> After making changes to the DDI file, you must update the database structure by using idb.exe (see Steps 4–7 in Section Adding Objects to Intellect.dbi).

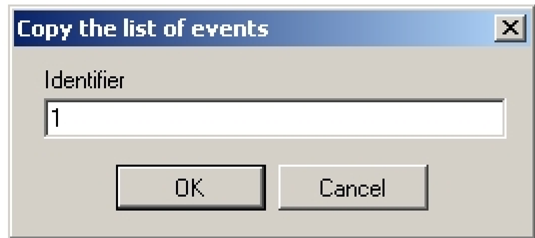
Additional Functionality of the ddi.exe Utility

The *ddi.exe* tool allows you to conveniently delete, add, and edit object properties (such as events and reactions), and copy them to the clipboard. In addition, you can copy object events to the clipboard, as a parameter of the *NotifyEvent* function. To do this, follow the steps below:

1. In the object list's context menu, select **Copy the list of events with NotifyEvent()**.



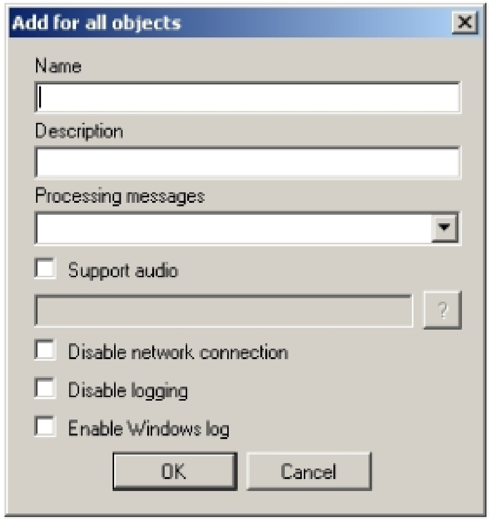
2. A dialog box opens. In the dialog box, enter the object ID to be used by the NotifyEvent function and click **OK.**



The event list is now copied. The clipboard contains object events in the format shown in the figure below.

```
NotifyEvent("CAM","1","ARM");
NotifyEvent("CAM","1","ATTACH");
NotifyEvent("CAM","1","BLINDING");
NotifyEvent("CAM","1","DETACH");
NotifyEvent("CAM","1","DISARM");
NotifyEvent("CAM","1","DISC_MOUNT");
NotifyEvent("CAM","1","DISC_UNMOUNT");
NotifyEvent("CAM","1","FILE_REC_ERROR");
NotifyEvent("CAM","1","MD_START");
NotifyEvent("CAM","1","MD_STOP");
NotifyEvent("CAM","1","PRINT");
NotifyEvent("CAM","1","REC");
NotifyEvent("CAM","1","REC_STOP");
NotifyEvent("CAM","1","RECORDER_OFF");
NotifyEvent("CAM","1","RECORDER_ON");
NotifyEvent("CAM","1","UNBLINDING");
```

To add an event for all objects, in the context menu, select **Add Event for All Objects**. The **Add for All Objects** dialog box opens. In the dialog box, specify the parameters of the new event.



To add objects from other DBI and DDI files, in the **File** menu, select **Insert from File**.

Creating MDL files

To create an MDL file, use two classes:

1. *NissObjectDLLExt*. All objects inherit from this class, whose virtual methods are redefined.
2. *CoreInterface*. The methods of this class are used to get parameters of the system's objects.

The declared classes and methods are contained in the nissdlle.h header file. The code contained in the nissdlle.h file is shown in APPENDIX 2. NissObjectDLLExt and CoreInterface class

declarations.

Note:
The methods of a class are the procedures and functions declared in its body.

The methods of the *NissObjectDLLExt* class are described in the table below.

<table>
<tr><th>Method</th><th>Description</th><th>Example</th></tr>
<tr><td>CoreInterface* m_pCore</td><td>A pointer to the core interface</td><td></td></tr>
<tr><td>virtual BOOL IsWantAllEvents()</td><td>Returns TRUE if the OnEvent function receives events from all objects; returns FALSE if the function receives events from its own object only.</td><td rowspan="2">If "CAM,GRABBER" is passed as a parameter, when settings of these objects are modified, the DEMO object receives the following messages:

DEMO|1|UPDATE_CAM|parameters of the camera

DEMO|1|UPDATE_GRABBER|parameters of the video capture card</td></tr>
<tr><td>virtual CString DescribeSubscribeObjectsList()</td><td>The method accepts a comma-separated list of objects. When an object from the list is modified, the current object is notified.</td></tr>
<tr><td>virtual CString GetObjectType()</td><td>Returns the object type</td><td>virtual CString GetObjectType()

{ return "DEMO"; }</td></tr>
<tr><td>virtual CString GetParentType()</td><td>Returns the parent object type</td><td>virtual CString GetParentType()

{ return "SLAVE"; }</td></tr>
</table>

virtual int GetPos()	Returns the position of the object in the *intellect.sec* key file.
	
	Attention! This parameter must be set in consultation with AxxonSoft.
		`virtual int GetPos()`
		
		`{ return -1; }`
		
		Note: If Intellect is run in the demo mode, the function returns ***-1***

virtual CString GetPort()	Returns the number of the port used for communication between the object and the core.
	
	Attention! This parameter must be set in consultation with AxxonSoft.
		`virtual CString GetPort()`
		
		`{ return "1100"; }`

virtual CString GetProcessName()	Returns the process name. Used by the core to search for and automatically run the executable module on startup of the system and initialization of the module	`virtual CString GetProcessName()`
		
		`{ return "demo"; }`

virtual CString GetDeviceType()	Determines the type of the object and its behavior. ACD – objects of this type receive all events related to the creation, modification, and deletion of the following objects: **Users**, **Time Zone**, and **Access Levels** ACD2 – a type similar to ACD, providing the additional (provided by the core) functionality of deleting temporary (fixed-term) cards The ACR type means that the object is a reader	All objects of the ACR type are available in the **Access Point** drop-down list
virtual BOOL HasChild()	Returns TRUE if the object has child objects, FALSE otherwise.	virtual BOOL HasChild() { return TRUE; }
virtual UINT HasSetupPanel()	Returns TRUE if the object has a setup panel, FALSE otherwise	virtual UINT HasSetupPanel() { return TRUE; }
virtual void OnPanelInit(CWnd*)	Used when the object's setup panel is initialized. The parameter is a pointer to the setup panel's window.	

virtual void OnPanelLoad(CWnd*,Msg&)	Used when the setup panel is loaded for setting the parameters of the object. The parameters are the setup panel's window and a message used to pass the parameters and fill in the relevant fields of the setup panel.	`virtual void OnPanelLoad(CWnd* pwnd,Msg& params)`
		`{`
		`CString s;`
		`s = arams.GetParam("port");`
		`pwnd->GetDlgItem(IDC_PORT)-> SetWindowText(s);`
		`}`

virtual void OnPanelSave(CWnd*,Msg&)	Used when the setup panel is saved for saving the parameters of the object. The parameters are a pointer to the setup panel's window and a reference to a message used to pass the parameters and save them in a database.	`virtual void OnPanelSave(CWnd* pwnd,Msg& params)`
		`{`
		`CString s;`
		`pwnd-> GetDlgItem(IDC_PORT)-> GetWindowText(s);`
		`params.SetParam("port",s);`
		`}`

virtual void OnPanelExit(CWnd*)	Used when the object's setup panel is closed ("exited"). The parameter is a pointer to the setup panel's window.	

virtual void OnPanelButtonPressed(CWnd*,UINT)	Used to handle clicks on the setup panel's buttons. The parameters are a pointer to the setup panel's window and a button ID.
	
	*Note: A button ID must be a number equal to or greater than **1151**. For example, the Resource.h file defines the ID of the **Test** button as follows:*
	
	#define IDC_TEST 1151
		Virtual void OnPanelButtonPressed (CWnd* pwnd,UINT id)
		
		{
		
		if(id==IDC_TEST)
		
		{ React react("DEMO",Id,"TEST");
		
		m_pCore->DoReact(react); }
		
		}

	If a button click is to open your own dialog box created in the same MDL file, you must first use the code shown in the example below.	HINSTANCE prev_hinst = AfxGetResourceHandle();
		
		HMODULE hRes = GetModuleHandle(“demo.mdl”);
		
		If (hRes) AfxSetResourceHandle (hRes);
		
		//Code for showing a dialog box:
		
		CXXXDialog dlg;
		
		dlg.DoModal();
		
		AfxSetResourceHandle(prev_hinst);

<table>
<tr><td>virtual BOOL IsRegionObject()</td><td>Shows whether the object supports Intellect's regions. Regions are used to group objects. They can also be used in the report system.</td><td></td></tr>
<tr><td>virtual BOOL IsProcessObject()</td><td>Shows whether the object supports starting and running multiple executable modules simultaneously. For example, this may be used for starting a separate module for each COM port.

Note: We recommend using one RUN file. This makes it easier to debug and modify the module.</td><td></td></tr>
<tr><td>virtual void OnCreate(Msg&)</td><td>Used when the object is created. The parameter is a reference to a message that contains object information. The method is also used to set default parameters.</td><td>virtual void OnCreate (Msg& msg)

{

msg.SetParam ("port","COM1");

}</td></tr>
<tr><td>virtual void OnInit(Msg&)</td><td>Used when the object is initialized. The parameter is a reference to a message that contains object information.</td><td>virtual void OnInit (Msg& msg)

{

OnChange (msg, msg);

}</td></tr>
</table>

virtual void OnChange(Msg&,Msg&)	Used when the object is changed. The first and second parameters are references to messages that contain object information before and after the change, respectively.	virtual void OnChange(Msg& msg, Msg& prev)
		{
		React react (msg.GetSourceType(), msg.GetSourceId(),"INIT");
		react.SetParam("port",msg.GetParam("port"));
		m_pCore->DoReact(react);
		}

virtual void OnDelete(Msg&)	Used when the object is deleted. The parameter is a reference to a message that contains object information.	virtual void OnDelete (Msg& msg)
		{
		React react (msg.GetSourceType(), msg.GetSourceId(),"EXIT");
		m_pCore-> DoReact(react);
		}

virtual void OnEnable(Msg&)	Used to handle clicks on the **Disable** button of Intellect's panel when the object is enabled. The parameter is a reference to a message that contains object information.	
virtual void OnDisable(Msg&)	Used to handle clicks on the **Disable** button of Intellect's panel when the object is disabled. The parameter is a reference to a message that contains object information.	
virtual BOOL OnEvent(Event&)		

	Used to handle the events that are passed as the parameter.	virtual BOOL
		
		OnEvent(Event& event)
		
		{
		
		If
		
		(event.GetAction() == "ACCESS_IN" ||
		
		event.GetAction() == "ACCESS_OUT")
		
		{
		
		Msg per = m_pCore-> FindPersonInfoByCard(event.GetParam("facility_code"),
		
		event.GetParam("card"));
		
		event.SetParam
		
		("param0", !per.GetSourceId().IsEmpty() ?
		
		per.GetParam("name") : event.GetParam("facility_code") + event.GetParam("card"));
		
		event.SetParam("param1", per.GetSourceId());
		
		}
		
		Else
		
		If (event.GetAction() == "NOACCESS_CARD")
		
		{
		
		event.SetParam

```
("param0",event.GetParam("facility_code") + event.GetParam("card"));

}
```

		``
		return TRUE;
		}
		``

virtual BOOL OnReact(React&)	Used to handle the reactions that are passed as the parameter.	

The *CreateNissObject(CoreInterface* core)* global function creates instances of the described objects, places them in an array (an instance of *CNissObjectDLLExtArray),* and returns a pointer to this array.   This function is used to receive a pointer to the core interface. This pointer is later used by objects to call interface methods:

```
CNissObjectDLLExtArray* APIENTRY CreateNissObject(CoreInterface* core)
{
 CNissObjectDLLExtArray* ar = new CNissObjectDLLExtArray;
 ar->Add(new NissObjectDemo(core));
 ar->Add(new NissObjectDemoDevice(core));
 return ar;
}
```

After loading a DDL file, the core calls the CreateNissObject function and receives pointers to all the objects in use.

All object setup panels are stored in resources as dialogs. Each dialog ID has the format **IDD_object_SETUP**, where **object** is the name of the corresponding object. For example, the ID of the **DEMO object** is **IDD_DEMO_SETUP**, and the ID of the **DEMO_DEVICE** object is **IDD_DEMO_DEVICE _SETUP.**

Note:
If you want for the settings tree to show a special icon for a particular object, in the resources of the DLL file, create a 14x14 **BITMAP** that contains the object name,.

MDL File Creation Wizard

To automate the creation of MDL files, use intellect_mdl.awx (see the *Wizard* folder in archive on page Intellect Software Integration Guide (HTTP API, IIDK, ActiveX)).

Use the Wizard to create an MDL file as follows:

1. Copy intellect_mdl.awx to the folder *Program Files\Microsoft Visual Studio\Common\MSDev98\Template*.
2. Start *Microsoft Visual C++.*
3. Create a new project with the name of **INTELLECT MDL WIZARD**.
4. Follow the instructions of the project configuration wizard.

As a result, a template for the system object is created. The project includes all of the necessary files, including a file that describes the object structure for intellect.dbi.

Attention!
In the project settings, change the output file extension from DLL to MDL.

Note:
For building the project, use **Release.**

Creating RUN files

Devices are managed by exchanging messages (commands) between RUN files and the system core. For implementing this interaction between software modules and the core, use the *Intellect Integration Developer Kit (IIDK),* which is covered in detail in Section Intellect Integration Developer Kit (IIDK). Other information is provided by the source files of the demonstration module; the files may be found in an appendix to this documentation.

Below is an example of how the *IIDK* is used in the *DEMO* module.

```
CString port = “1100”;

CString ip = “127.0.0.1”;

CString id = “”;

BOOL IsConnect = Connect (ip, port, id, myfunc);

if (!IsConnect)

{

// connection failed

AfxMessageBox("Error");

Return;

}

SendMsg(id,”CAM|1|REC”); // turn on recording for camera 1

SendMsg(id, "DEMO|1|RESTORE"); // restore the connection with the DEMO object

//turn on the DEMO_DEVICE with address 1

SendMsg(id,"DEMO_DEVICE|1|ON|params<1>,param0_name<address>,param0_val<1>");

Disconnect(id);
```

Attention!
If an MDL file exists, connecting to Intellect's core does not require creating an IIDK Interface object in the system. The connection ID passed is an empty string (in other words, the ID is "").

When a module is unloaded, it receives the **WM_EXIT** event:

#define WM_EXIT (WM_USER+2000)

Use a WinAPI function, *PostThreadMessage,* to catch this message and ensure that the module is unloaded properly. In *VC++* and *MFC,* the **WM_EXIT** event is caught in a subclass of *CWinApp*; in *Delphi* and *CBuilder,* it is caught in a subclass of *TApplication*.

Creating and Configuring Integrated Objects (Modules) in Intellect

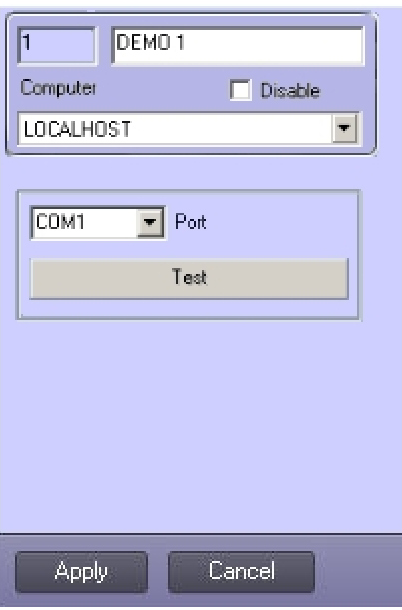
To create and configure an integrated object (module) in Intellect:

1. Copy the MDL and RUN files to the *Intellect\Modules* folder.
2. Start Intellect.
3. Under the **Computer** object, create the objects added earlier using the software module. For the *DEMO* module, Under the **Computer** object, create the **DEMO** object.

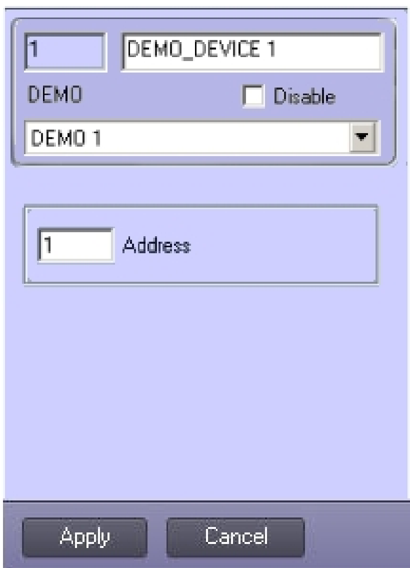
Note:
Under the **DEMO** object, create a child object, **DEMO_DEVICE.**

As a result, the setup panels of the created objects become available.

DEMO object setup panel:



DEMO_DEVICE object setup panel:



4. Set up the objects.

The integrated objects are now created and set up in Intellect.

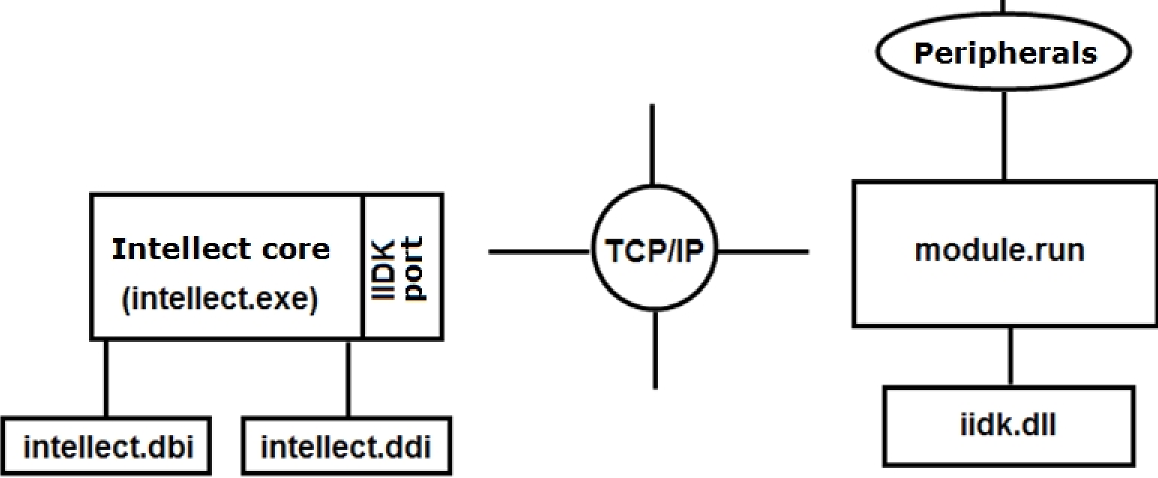
Intellect Integration Developer Kit (IIDK)

General Information on IIDK

Purpose of the IIDK

System expandability is supported by Intellect's software architecture. Expandability allows communication between the core and functional modules (third party information systems) via TCP/IP.

The figure below shows a diagram of interaction between Intellect's core and external software (a functional module).



Interaction is done by exchanging messages in a communication environment; message exchange is implemented by using the *IIDK*.

The *Intellect Integration Developer Kit* (*IIDK*) is a set of development tools for integrating third-party security software into Intellect. This kit allows you to expand the system rapidly and effectively by adding functional modules that support new hardware and new functions.

Developer Requirements

To use the *IIDK*, you must:

1. know how to program in C/C++;
2. know the basics of Win32 programming;
3. have an IDE with DLL support (such as *Microsoft Visual C++*, *C++ Builder,* or *DELPHI*).

Note:
When creating LIB files with C++ Builder 5's implib.exe tool, add the "**- a**" option.

IIDK Components

The *IIDK* includes the following development tools:

1. *iidk.ocx* – ActiveX control. When installing *Intellect* this file is stored in the Windows\System32 folder and registered in OS.
2. *ddi.exe* – tool used for viewing and editing DDI- and DBI- files. It is stored in the <*Intellect* installation directory>\Tools folder.

Connecting to Intellect

Connection Parameters

Intellect's core interacts with functional modules (third party information systems) according to the following connection parameters:

1. Port number.
    a. For the video subsystem: 900.
    b. For the **Interface IIDK** object: 1030.
    c. For **ATM** objects: 1009.

    Note.
    1030 (IIDK) port can be in use to connect ATMs (ATM) (not only 1009 port) - in this case the **ATM** object will be marked with red cross in the hardware tree. For this the **IIDK Interface object is to be created in the hardware tree**.

2. The IP address of the computer that is running Intellect's core.
3. ID (the connection object ID).

Attention!
To connect to video subsystem (port 900) id is to be more than 1 and must not be the same as id of **IIDK Interface objects created in the system**. To connect to **IIDK Interface object** (port 1030) id is to be the same as one of the object specified in the dialog box of *Intellect* settings.

Note.
If connection to the server (**IIDK Interface object**) from remote computer is required, then there is no need to install *Intellect* on the remote computer, but this computer is to be added to *Intellect* configuration on the server (in the **Hardware** tab of the **System settings** dialog box) and **IIDK Interface object** is to be created under the created **Computer** object. In this case the server address is to be specified in IP parameter of Connect function and ID of specified **IIDK Interface object** is to be specified in ID parameter. Take into account the fact that the **Computer** object corresponding to the remote computer is marked with a red cross in the object tree.

Note:
If an MDL file exists (see Section Creating MDL files), the connection to Intellect's core does not require creating the **IIDK Interface** object in the system. The connection ID passed is an empty string (in other words, the ID is "").

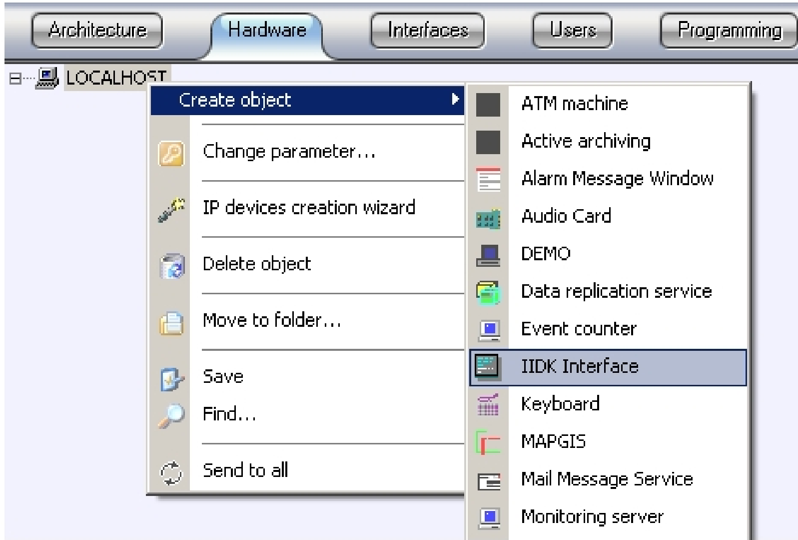
IIDK Interface Object

With the **IIDK Interface** object one can manage all the elements of the system. The **IIDK Interface** object is created under the **Computer** object in the *Intellect* object tree.

Note
To use the **IIDK Interface** object, allow the relevant functionality in the license key.

Note
If *Intellect* is started in the demo mode, the **IIDK Interface** object is activated after the functional module is connected to the system core (see Connect section)

Important!
The ID of the **IIDK Interface** object must not be the same as the IDs of the **Monitor** objects created in the system.



If the **IIDK Interface** object is used, the settings panels are not created for integrated functional modules (third party applications).

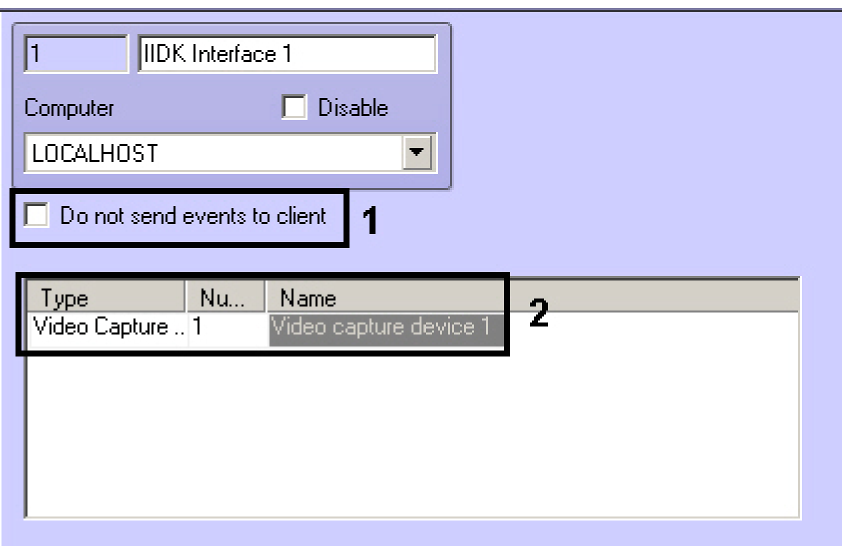
When the *Intellect* distributed architecture is used, the **IIDK Interface** object must be created on the computer that is running the software core (the core to which the connection is made). If the connection is made to a computer that has the *Operator Workstation* installed, the connection parameters must include the IP address of the *Server* or the *Administrator Workstation*.

Configuring passing events throung IIDK Interface object

IIDK Interface allows configuring of event filtering passed to connected client applications.

To configure filtering, do the following:

1. Go to the settings panel of the created **IIDK Interface** object.

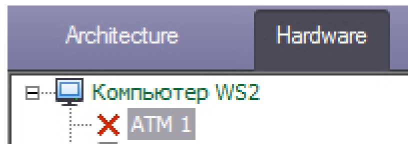


2. Set the **Do not send events to client** if it is not required to send any messages to client application (1).
3. In the table (2) specify list of objects, events from which have to be sent to connected client applications.

Configuring of event filtering is completed.

Features of ATMs integration. ATM object

The **ATM** object can be used to send events from ATM software to *Intellect* core. This object is created under the **LOCALHOST** object in the **Hardware** tab of the **System settings** dialog box. It is created instead of the **IIDK Interface** object.



The **ATM** object shows ATM events (“Card inserted”, “Withdraw card”, etc.) in the event viewer. These events can be used for captioning, reactions configuration, etc.

Note.
For events regarding client card masked bank card number can be sent in the param0 parameter.

The list of available **ATM** events can be seen using the ddi.exe utility. Information on how to use this utility can be found in the The ddi.exe utility for editing database templates and external settings files section of Administrator's Guide.

Connection method and message syntax for the **ATM** object are the same as those for the **IIDK Interface object**, though 1009 port is in use for sending messages (see also Connection Parameters and Message Syntax (port 900)).

IIDK Functions

Connect

To establish communication between a functional module and Intellect, connect to the system core by using the following function:

```
BOOL Connect (LPCTSTR ip, LPCTSTR port, LPCTSTR id, void (_stdcall *func)(LPCTSTR msg))
```

Parameters of the connection function:

Parameter	Description	Example
LPCTSTR ip	The ID address of the computer that is running the system core	```CString port = "900"; CString ip = "127.0.0.1"; CString id = "2"; BOOL IsConnect = Connect(ip, port, id, myfunc); if (!IsConnect) { // connection failed AfxMessageBox("Error"); }```
LPCTSTR port	TCP/IP connection port	
LPCTSTR id	A connection ID, for video	
_stdcall *func)(LPCTSTR msg))	A callback function that accepts messages from Intellect	

Example code (from the table above):

```
CString port = “900”;

CString ip = “127.0.0.1”;

CString id = “2”;

BOOL IsConnect = Connect(ip, port, id, myfunc);

if (!IsConnect)

{

  // connection failed

  AfxMessageBox("Error");

}
```

The function returns TRUE if the connection is established, or FALSE if not.

All messages from the system core are accepted by a callback function.

A sample declaration of the callback function:

```
void _stdcall myfunc(LPCTSTR str)
{
 printf("\r\nReceived:%s\r\n\r\n",str);
}
```

> **Note:**
> **Void _stdcall myfunc** is called in a separate stream (not in the application's main stream).

The developer handles received messages as needed.

SendMsg

To send messages to the system core, use the following function:

```
BOOL SendMsg (LPCTSTR id, LPCTSTR msg)
```

Parameters of the SendMsg function:

Parameter	Description	Example

LPCTSTR id	The connection ID passed in the call to the **Connect** function	CString port = “900”;
		
		CString ip = “127.0.0.1”;
		
		CString id = “2”;
		
		BOOL IsConnect = Connect(ip, port, id, myfunc);
		
		if (!IsConnect)
		
		{
		
		// connection failed
		
		AfxMessageBox("Error");
		
		Return;

LPCTSTR msg	Message text	}
		
		 SendMsg(id,"CAM|1|REC"); // turn on recording for camera 1
		
		 Disconnect (id);

The function returns TRUE if the message was sent, otherwise FALSE

Disconnect

To terminate a connection, use the **Disconnect** function:

```
void Disconnect (LPCTSTR id)
```

, where **LPCTSTR id** is the connection ID passed in the call to the **Connect** function.

If the connection is terminated by Intellect, **DISCONNECTED** is passed to the callback function.

Note:
An example of using the **Disconnect** function is given in Section SendMsg.

Other functions

On the page:

- Connect3
- SendReactToCore
- IsConnected
- Connect4
- SendData4
- SendFile
- GetMsg

The iidk.h header file contains extra functions that are listed below. The Connect4, SendData4, SendFile and GetMsg functions should not be used. They are created for internal use. The Connect2 function is not used.

Connect3

```
BOOL Connect3(LPCTSTR ip, LPCTSTR port, LPCTSTR id, iidk_callback_func* lpfunc,
            DWORD user_param,int async_connect,DWORD connect_attempts)
```

Parameter	Description
ip	IP address of *Intellect* Server
port	TCP/IP port over which the connection is established
id	Slave connection ID, for video
lpfunc	Callback function receiving messages from *Intellect*
user_param	Extra parameter that comes to Callback function in order to split slaves if there is only one function.
async_connect	0 - synchronous connection mode, the function returns TRUE if the connection is established. -1 - asynchronous connection mode, the function always returns FALSE if the connection is established, then the CONNECTED event is created. Any other value – at first the synchronous mode is used, in case of fault - asynchronous mode.
connect_attempts	Number of connection attempts.

SendReactToCore

The function is used to send a reaction to the specific core.

```
BOOL SendReactToCore(LPCTSTR id, LPCTSTR msg)
```

Parameter	Description
id	Core connection ID
msg	Messages sent. Message format is similar to SendMsg.

IsConnected

IsConnected returns TRUE if the client is connected to server.

```
BOOL IsConnected();
```

Connect4

```
BOOL Connect4(LPCTSTR ip, LPCTSTR port, LPCTSTR id,  iidk_callback_func* lpfunc,
            iidk_frame_callback_func* lpframe_func, iidk_user_data_func* iidk_user_data_func,
            DWORD user_param,int async_connect,DWORD connect_attempts);
```

Parameter	Description
ip	IP address of *Intellect* Server
port	TCP/IP port over which the connection is *established*
id	Core connection ID, for video
lpfunc	Callback function receiving messages from *Intellect*
lpframe_func	Callback function receiving video frames
iidk_user_data_func	Callback function for data sent using the SendData4 function
user_param	Extra parameter that comes to the Callback function in order to split slaves if there is only one Callback function for all cores.
async_connect	0 - synchronous connection mode, the function returns TRUE if the connection is established -1 - asynchronous connection mode, the function always returns FALSE. If the connection is established, then the CONNECTED event is created Any other value – at first the synchronous mode is used, in case of fault - asynchronous mode.
connect_attempts	Number of connection attempts

SendData4

This function is used to send CUserNetObject. Its purpose is to send raw data.

```
BOOL SendData4(LPCTSTR id, int nIdent,BYTE *pBuffer,DWORD dwSize);
```

Parameter	Description
id	Core connection ID
nIdent	Data UID
pBuffer	Transmitted data

dwSize	The size of data array

SendFile

The function is used to send a file.

```
BOOL SendFile(LPCTSTR id, LPCTSTR file_from, LPCTSTR file_to)
```

Parameter	Description
id	Core connection ID
file_from	Address to send file from.
file_to	Address to send file to.

GetMsg

The function is used to retrieve incoming messages that are queued if the Callback function is not specified.

```
BOOL GetMsg(LPTSTR msg, DWORD& cb)
```

Parameter	Description
msg	Incoming message
cb	Message length

Sent Message Syntax

Message Syntax

Messages sent to the core have the following syntax:

CORE||DO_REACT|source_type<OBJECT TYPE>,source_id<OBJECT ID>,action<ACTION> [,params<NO. OF PARAMETERS>,param0_name<PARAMETER NAME_0>,param0_val<PARAMETER VALUE_0>]

Below is the syntax of messages that contain two parameters.

CORE||DO_REACT|source_type<OBJECT TYPE>,source_id<OBJECT ID>,action<ACTION>,params<2>,param0_name<PARAMETER NAME_0>,param0_val< PARAMETER VALUE_0>,param1_name<PARAMETER NAME_1>,param1_val<PARAMETER VALUE_1>

The message parameters are described in the table below.

Parameter	Description
source_type<obj>	Object type (see the DDI file ([OBJTYPE] section))
source_id<id>	The object ID set when creating the object in Intellect (see Intellect's settings tree)
action<react>	Action (see the DDI file (the [REACT] section))
params<number>	The number of parameters passed, in decimal format
param0_name<str1>	Parameter name
param0_val<str2>	Parameter value

Note:
For working with DDI files, we recommend using the ddi.exe utility (see Section Using the ddi.exe Tool to Work with DDI files).

Example. Sending a message to switch the telemetry to preset mode 4.

CString msg=

"CORE||DO_REACT|source_type<TELEMETRY>,source_id<1.1>,action<GO_PRESET>,params<2>,param0_name<preset>,param0_val<4>,param1_name<*tel_prior*>,param1_val<2>";

SendMsg(id,msg);

Message Syntax (port 900)

Messages sent to port 900 are passed to the video subsystem directly; for this reason, such messages have a different syntax.

Messages sent to the video subsystem have the following syntax:

OBJECT TYPE|OBJECT ID|ACTION [|PARAMETER<VALUE>]

Below is the syntax of messages that contain n parameters.

OBJECT TYPE|OBJECT ID|ACTION [|PARAMETER 1<VALUE>,PARAMETER 2<VALUE>,...,PARAMETER N<VALUE>]

Attention!
Port 900 may only be used to manage objects of the GRABBER, CAM, or MONITOR types.

The message parameters are described in the table below.

Parameter	Description
Object type	Object type (GRABBER, CAM, or MONITOR)
Object ID	The object ID set during creation of the object in Intellect
Action	Action (command)

Parameter <Value>	Parameter name. The value is enclosed by angle brackets.

Example 1. Setting camera 1 to recording mode.

CString msg = “CAM|1|REC”;

SendMsg (id,msg);

Example 2. Saving video from all cameras to local disk C.

CString msg = “GRABBER|1|SET_DRIVES|drives<C:\>” ;

SendMsg(id,msg);

Note.
The **SET_DRIVES** command includes the ID of any of the video capture cards created in the system.

Note.
The **SET_DRIVES** command does not change the video archiving settings set in the system.

Using the Event and React classes

For working with messages, you may use the classes provided: *Event* and *React,* declared in the msg.h file.

Example of using the react class:

A message composed without using the classes	A message composed using the React class
CString msg = “CORE||DO_REACT|source_type<TELEMETRY>,source_id<1.1>, action<GO_PRESET>,params<2>,param0_name<preset>,param0_val<4>, **param1_name<*tel_prior*>,param1_val<2>”;** **SendMsg(id,msg);**	**React react(“TELEMETRY”,”1.1”,”GO_PRESET”);** **react.SetParamInt(“preset”,4);** **react.SetParamInt(“*tel_prior*”,2);** **SendMsg(id,react.MsgToString().c_str());**

Note:
The msg.h and msg.cpp files are located in the Misc folder which is in the archive on page Intellect Software Integration Guide (HTTP API, IIDK, ActiveX).

Examples of Managing System Objects

Adding, Updating, and Deleting System Objects

On the page:

- Adding a User to a Department
- Adding and Deleting a Video Capture Card

System objects are added, updated, and deleted by the following commands:

1. **CORE||CREATE_OBJECT** – creates a new object.
2. **CORE||UPDATE_OBJECT** – updates an existing object or creates a new one.
3. **CORE||DELETE_OBJECT** – deletes an object.

Adding a User to a Department

Below is a message that adds the specified user to the specified department, with the specified parameters:

CORE||CREATE_OBJECT|objtype<PERSON>,objid<12>,parent_id<1>,name<John Doe>,core_global<0>,params<11>,param0_name<facility_code>,param0_val<122>,param1_name<card>,param1_val<1234>,param2_name<pin>,param2_val<>,param3_name<comment>,param3_val<HR Department Head>,param4_name<is_locked>,param4_val<0>,param5_name<is_apb>,param5_val<0>,param6_name<level_id>,param6_val<*>,param7_name<person>,param7_val<>,param8_name<_creator>,param8_val<1>,param9_name<expired>,param9_val<>,param10_name<temp_card>,param10_val<>

Adding and Deleting a Video Capture Card

If an object is not present in the system, add that object with the **UPDATE_OBJECT** command (the system must not have an object with a type and ID equal to objtype and objid, respectively).

CORE||UPDATE_OBJECT|objtype<GRABBER>,objid<12>,core_global<0>,parent_id<SLAVAXP>,name<Frame grabber 1>,params<5>,param0_name<format>,param0_val<NTSC>,param1_name<mode>,param1_val<1>,param2_name<chan>,param2_val<2>,param3_name<type>,param3_val<FX 4>,param4_name<resolution>,param4_val<0>

Having received the following message, the system changes the name of an existing object:

CORE||UPDATE_OBJECT|objtype<GRABBER>,objid<12>,core_global<0>,parent_id<SLAVAXP>,name<Card 2>

To delete an object and all of its child objects, use the **DELETE_OBJECT** command:

CORE||DELETE_OBJECT|objtype<GRABBER>,objid<12>

Working with the System in the Multiuser Mode

The remote computer must install and be running Intellect (**Client** installation version) in order to exchange messages with the Server.

If users have been created and access rights have been configured in Intellect, any message that requires a response from the system core must contain the **receiver_id<ID>** parameter, where ID is the ID of the **IIDK Interface** in the system.

CORE||GET_CONFIG|objtype<CAM>,objid<1>,receiver_id<1>

// Returns the parameters of the Camera 1 object

Determining Computers Where Intellect was Unloaded (via Port 1030)

If Intellect is unloaded, the callback function receives a message with an **action** parameter value of **DISCONNECTED**:

ACTIVEX|12|EVENT|SOCKET<>,MMF<>,objaction<DISCONNECTED>,TRANSPORT_TYPE<MMF>,core_global<1>, action<DISCONNECTED>, module<slave.exe>, objtype<SLAVE>,__slave_id<SLAVAXP.12>, objid<SLAVAXP>,owner<SLAVAXP>,TRANSPORT_ID<1111>,time<12:41:16>,date<23-09-02>

The message contains the name of the computer on which *Intellect* was unloaded and the date and time

Redirecting Video Cameras to the Monitor

After receiving the following message, the system deletes all cameras from the monitor and calls the specified video camera:

CORE||DO_REACT|source_type<MONITOR>,source_id<1>,action<REPLACE>,params<4>,param0_name<slave_id>,param0_val<SLAVA>,param1_name<cam>,param1_val<1>,param2_name<control>,param2_val<1>,param3_name<name>,param3_val<>

If connected via port 900, the above action is performed by using the following message:

MONITOR|1|REPLACE|slave_id<SLAVA>,cam<1>,control<1>

Obtaining Object Parameters (via Port 1030) GET_CONFIG

An example of use of the **GET_CONFIG** command is given below:

CORE||GET_CONFIG|objtype<CAM>,objid<1>

The returned message contains all the parameters of the specified object:

ACTIVEX|12|OBJECT_CONFIG|rec_priority<0>,mask0<>,decoder<0>,mask1<>,flags<>,mask2<>,compression<3>,sat_u<5>,mask3<>,proc_time<>,hot_rec_period<>,mask4<>,telemetry_id<>,manual<1>,region_id<1.1>,contrast<5>,md_mode<0>,md_size<5>,audio_type<>,pre_rec_time<0>,config_id<>,bright<7>,alarm_rec<0>,audio_id<>,rec_time<>,hot_rec_time<2>,activity<>,mux<0>,parent_id<1>,objtype<CAM>,type<>,__slave_id<SLAVAXP.12>,objid<1>,name<Camera 1>,objname<Camera 1>,color<1>,priority<0>,md_contrast<5>

Note:
To obtain the configuration of all the objects of the specified type, remove the **objid** parameter.

Example. Get information about user by the identifier.

CORE||GET_CONFIG|objtype<PERSON>,objid<1>

The response is the message with parameters containing necessary information, including user name, card number etc.:

ACTIVEX|1|OBJECT_CONFIG|pnet3_sound<0>,galaxy_dual_focus<0>,auto_pass_type<>,galaxy_pin_change<0>,external_id<>,card_date<26.05.2017 10:57:06>,galaxy_tag_link<0>,rubeg8_zone_id<>,levels_times<>,expired<>,hid_escort_id<>,objtype<PERSON>,level2_id<>,galaxy_group_choice<0>, who_level<>,hid_use_extended_access<0>,visit_purpose<>,**card<1234>**,email<>,galaxy_timer_schedule<0>,galaxy_menu_option<0>,aiu_holiday<0>, area_id<>,aiu_alarm<0>,objname<User 1>,surname<>,who_card<>,auto_brand<>,pnet3_alarm<0>,card_loss<0>, **facility_code<432>**,galaxy_temp_code<0>,post<>,when_area_id_changed<>,drivers_licence<>,bolid_in_device<0>,pnet3_acs_counter<0>, temp_levels_times<>,temp_card<>,pnet3_no_entry<0>,location<>,temp_level_id<>,patronymic<>,teleph_work<>,department<>,galaxy_keypad<0>, _TRANSPORT_ID<>,finished_at<>,aiu_ksd_type<>,all_param<>,galaxy_template<0>,tabnum<>,parent_id<1>,pur<>,galaxy_duress<0>,pnet3_no_exit<0>, galaxy_dual<0>,hid_pin_exempt<0>,pnet3_counter<0>,whence<>,schedule_id<>,hid_enable_pin_commands<0>,galaxy_dual_access<0>, pnet3_block<0>,passport<>,person<>,galaxy_menu_choice<0>,flags<0>,auto_number<>,phone<>,pin<>,rubeg8_AccessToBCPTimeZoneNumber<>, begin_temp_level<>,pnet3_master<0>,aiu_mark<0>,end_temp_level<>,visit_birthplace<>,galaxy_menu_level<>,visit_document<>,pnet3_guard<0>, pnet3_black<0>,aiu_kso_type<>,is_apb<0>,**name<User 1>**,pnet3_temp<0>,started_at<>,level_id<>,_marker<>, bolid_user_type<>,owner_person_id<>,card_type<>,is_active_temp_level<0>,begin<>,hid_line_tag<>,guid<{5B358685-E041-E711-BCC6-DA0AE28E0C17}>, pnet3_guest<0>,is_locked<0>,objid<1>,marketing_info<>,comment<>,aiu_vpu_arm<0>,visit_reg<>,bolid_in_pku<0>,pnet3_2cards_mask<0>

Note.
User data can also be received via direct request to *Intellect* software database from the OBJ_PERSON table. In this case you can select a user by card number or other parameters. See more info on *Intellect* software database operation in Administrator's Guide, the Appendix 4. INTELLECT™ software database management.

Obtaining Information on Object States GET_STATE and GET_LIST

To obtain information on the state of an object, use the **GET_STATE** command:

CORE||GET_STATE|objtype<CAM>,objid<1>

The following string is returned:

ACTIVEX|12|OBJECT_STATE|objtype<CAM>,__slave_id<SLAVAXP.12>,objid<1>,state<DISARM_DETACHED>

The state of the specified object is represented by the **state** parameter, which takes values from the set of states that are specified in the object's DDI file.

If connected via port 900, requests for object states are performed through the **GET_LIST** command:

CAM||GET_LIST

Note:
Regardless of whether an object ID is specified, the command returns the states of all objects of the specified type.

The returned message has the following format:

CAM|1|SETUP|rec_priority<0>,is_armed<0>,is_recorded<0> , bt<0> , slave_id<SLAVAXP>, compression<3>,sat_u<5>, proc_time<0>, hot_rec_period<0> , manual<1>, telemetry_id<>, is_detached<1> , contrast<5>, md_size<5>,md_mode<0>, is_alarmed<0>, audio_type<>, pre_rec_time<0>, bright<7>, audio_id<>, rec_time<0>, alarm_rec<0>, hot_rec_time<2>, mux<0>, parent_id<1>, __slave_id<SLAVAXP>, priority<0>, mask<>, color<1>,md_contrast<5>, is_ring<1>

The message presents the states as follows: **is_state<val>,** where **state** is an object state (see the DDI file); and **val** equals 1 if the object is in this state, 0 otherwise.

Note.
The **is_ring<>** parameter shows whether loop recording is performing or not.

Showing Information Messages. SET_STATE

To show an information message on the display of Intellect's main control panel, use the SET_STATE command:

CORE||SET_STATE|name<POS 1>,value<Can’t open port COM4>

The figure below shows the result of handling the message by the system.

The message is removed from the display as follows:

CORE||SET_STATE|name<POS 1>,value<>

Live and archived video

To get live video from Camera 1 send **CAM|1|START_VIDEO|compress<1>** message to port 900.
Here compress<> is compression ratio, from 0 to 5. Video frames will be received as a response to this message. The example of how to process incoming frames is given in demo kit available for download at the page of Intellect Software Integration Guide (HTTP API, IIDK, ActiveX).

To get archived video from Camera 1 send **CAM|1|ARCH_FRAME_TIME|time<dd-mm-yy HH:MM:SS.FFF> (to specify start time for viewing the archive)** or **CAM|1|PLAY|compress<> (to get archived video**. **Archived video is handled the same way as live video)** messages to port 900.

In order to get the full list of time intervals with video recordings for exact date, send **CAM|id|ARCH_GET_INTERVALSREC|date<>** message to port 900.

The date<> parameter can take the date<dd-mm-yy|> value or it can be left blank. In the first case time intervals for specified date will be requested, in the second – dates for which there is an archive.

As a result the **Event: CAM|id|SET_INTERVALSREC|intervals<>,date<>** message is received.

The value of intervals<> parameter looks like this: intervals<begin1 end1\nbegin2 end2…\nbeginN endN|date1\ndate2…\ndateN\n>

The time of beginning and ending are one blank separated (0x20 code), intervals are line break separated '\n'(0x0A code).

- begin1, begin2, ... beginN – time of interval beginnings in the HH:MM:SS format (returns if the exact date was requested).
- end1, end2, ... endN – time of interval endings in the HH:MM:SS format (returns if the exact date was requested).
- date1, date2, ... dateN – dates at which there are recordings in the archive (returns if the date field in the request is blank or there is no such field).

date<dd-mm-yy|> parameter represents date for which the intervals were requested or blank value (date<>) if dates for the entire period were requested.

Telemetry control via IIDK

Telemetry is controlled via IIDK using simple reactions described in the TELEMETRY section of Programming guide, for instance:

CORE||DO_REACT|source_type<TELEMETRY>,source_id<1.1>,action<LEFT>,params<1>,param0_name<tel_prior>,param0_val<3> – message sent to port 1030 in order to rotate camera lens left with high priority.

TELEMETRY|1.1|LEFT|speed<2>,tel_prior<3> – reaction to port 1030 in order to rotate camera lens left with high priority at an average speed.

Map layer operations

The command for setting the parameter and position of **Camera 1** object icon is run in one of the following ways:

1. Sending a message to port 1030 **CORE||DO_REACT|source_type<MAPLAYER>,source_id<1>,action<CUSTOMIZE_OBJECT>,params<7>,param0_name<x>,param0_val<200>,param1_name<y>,param1_val<200>,param2_name<objtype>,param2_val<CAM>,param3_name<objid>,param3_val<1>,param4_name<a>,param4_val<90>,param5_name<w>,param5_val<70>,param6_name<h>,param6_val<80>**
   Where *x*, *y*, *w* and *h* are the coordinates and size of the object icon on the map.
   a is a tilt angle of icon.
2. Sending a reaction to port 1030
   MAPLAYER|1|CUSTOMIZE_OBJECT|x<200>,y<200>,objtype<CAM>,objid<1>,a<90>,w<70>,h<80>

Layer 1 is shown in the interactive map window using one of the following ways:

1. Sending a message to port 1030: **CORE||DO_REACT|source_type<MAPLAYER>,source_id<1>,action<ACTIVATE>**
2. Sending a reaction to port 1030: **MAPLAYER|1|ACTIVATE**

CamMonitor.ocx ActiveX Control

General description of CamMonitor.ocx component of ActiveX

On the page:

- General information
- Requirements to developers

General information

CamMonitor.ocx is the component of ActiveX that is similar in every way to the **Video monitor** interface object. It allows you to manage cameras, view the archive, etc.

CamMonitor.ocx component supports operation in the demo mode.

Requirements to developers

To use CamMonitor.ocx you will need:

1. The knowledge of any programming language that supports using the Component Object Model (COM);
2. Basic knowledge of Win32/Win64 programming;
3. Programming environment that supports OCX files.

Requirements for software which is used while integrating

How to install CamMonitor.ocx

Important!
It is not recommended to install *Intellect* and CamMonitor.ocx on the same computer. If it is required, then their versions are to be coincident, otherwise CamMonitor.ocx operation is not guaranteed.

Use the CamMonitorInstaller.exe file (stored in the <*Intellect* installation directory>\Redist\CamMonitor folder) to install CamMonitor.ocx. Both 32-bit and 64-bit versions of this component are available. The installation files of components of different bit count are stored in the corresponding folders (x86 and x32 correspondingly).

If *Intellect* software is installed on the computer, then the CamMonitor.ocx file is stored in the <*Intellect* installation directory>\Modules\ folder when installing the 32-bit version and it is stored in the <*Intellect* installation directory>\Modules64\ folder when installing the 64-bit version.

If *Intellect* software is not installed, then the 32-bit component is installed in the c:\Program Files (x86)\ITV VideoPlayer\Modules\ folder and the 64-bit component is installed in the c:\Program Files\ITV VideoPlayer x64\Modules64\ folder.

There is standard registration of CamMonitor.ocx as ActiveX component at the stage of installation.

CamMonitorInstaller.exe installs the required files for all users.

Besides the library itself, CodecPack driver pack and ITV VideoPlayer utility are installed. ITV VideoPlayer utility uses the CamMonitor.ocx component and enables viewing the archive

from the selected camera. By default this utility is installed in the C:\Program Files\ITV VideoPlayer\ folder. The utility interface looks like one of Converter.exe, but some features of Converter.exe (not dealing with viewing the archive) are not available. This utility can be used to check if CamMonitor.ocx is installed correctly.

CamMonitor.ocx parameters

On the page:

- CamMenuOptions
- CamMenuProcessingOptions
- CamButtonsOptions
- MainPanelOptions
- KeysOptions
- OverlayMode
- How to use parameters

The parameters used for setting the CamMonitor component are presented in this chapter: elements to be shown as well as the overlay mode.

All parameters are *long*-like integers.

The values of parameters used for interface setup are enlisted in the tables and formed in a way that one integer only is represented in binary notation. To set the value of a parameter, combine the values of parameters using the XOR operation. You'll get the number the positions of which in binary notation represent the interface elements that are to be shown and those to be hidden. See How to use parameters section.

The OverlayMode parameter differs from others: it possesses values from 0 to 2 and its value sets the overlay mode.

CamMenuOptions

CamMenuOptions : long

Allows setting the feature menu of the camera.

One or more checkboxes can be set checked.

Available values:

Value	**Information**
#define MENU_ENABLE_OPTION 0x00000001	Show the menu
#define MENU_ARM_ENABLE_OPTION 0x00000002	Show the **Arm** option
#define MENU_REC_ENABLE_OPTION 0x00000004	Show the **Start recording** option
#define MENU_CAMS_ENABLE_OPTION 0x00000008	Show the **Camera** option
#define MENU_TITLES_ENABLE_OPTION 0x00000010	Show the **Show titles** option
#define MENU_PROCESSING_ENABLE_OPTION 0x00000020	Show the **Processing** option
#define MENU_EXPORT_ENABLE_OPTION 0x00000040	Show the **Export** option

CamMenuProcessingOptions

CamMenuProcessingOptions : long

Allows setting the **Processing** menu in the feature menu of the camera.

One or more checkboxes can be set checked:

Available values:

Value	**Information**
#define MENU_PROCESSING_DEINTERLACE_ENABLE_OPTION 0x00000001	Show the **Deinterlacing** option
#define MENU_PROCESSING_ZOOM_ENABLE_OPTION 0x00000002	Show the **Zoom-in** option
#define MENU_PROCESSING_CONTRAST_ENABLE_OPTION 0x00000004	Show the **Contrast** option
#define MENU_PROCESSING_MASK_ENABLE_OPTION 0x00000008	Show the **Detector mask** option

CamButtonsOptions

CamButtonsOptions : long

Set up displaying the buttons of the CamMonitor component.

One or more checkboxes can be set checked.

Available values:

Value	**Information**
#define BUTTON_ARCH_ENABLE_OPTION 0x00000001	Show the Archive button
#define BUTTON_TIME_ENABLE_OPTION 0x00000002	Show time
#define BUTTON_NAME_ENABLE_OPTION 0x00000004	Show camera name
#define BUTTON_MENU_ENABLE_OPTION 0x00000008	Show the Menu button
#define BUTTON_RAYS_ENABLE_OPTION 0x00000010	Not used
#define BUTTON_MICS_ENABLE_OPTION 0x00000020	Not used

MainPanelOptions

MainPanelOptions : long

Set up displaying the CamMonitor panel.

One or more checkboxes can be set checked.

Available values:

Value	Information
#define MAIN_PANEL_ENABLE_OPTION 0x00000001	Show the panel

KeysOptions

KeysOptions : long

It allows setting control over the component using the keyboard and mouse.

One or more checkboxes can be set checked.

Available values:

Value	Description
#define TELEMETRY_DISABLE_OPTION 0x00000001	Disables control over the CamMonitor component using the hotkeys available for Video Monitor (see Video Monitor).
#define TELEMETRY_DISABLE_OPTION 0x00000002	Disables Telemetry control using the CamMonitor component (see Telemetry control).

OverlayMode

OverlayMode : long

Set the overlay mode.

Available values:

Value	Information
0	Overlay is not in use
1	Overlay 1
2	Overlay 2

How to use parameters

```
DWORD options = CamMonitor1->CamMenuOptions;
options = options^MENU_CAMS_ENABLE_OPTION^MENU_ARM_ENABLE_OPTION^MENU_REC_ENABLE_OPTION;
CamMonitor1->CamMenuOptions = options;
CamMonitor1->CamMenuProcessingOptions ^= MENU_PROCESSING_MASK_ENABLE_OPTION;
```

CamMonitor.ocx methods

On the page:

- Connect
- ShowCam
- DoReactMonitor
- RemoveAllCams
- IsConnected
- GetCurIp
- SendRawMessage
- Disconnect
- SetCallBackOptions

Connect

Connect(BSTR **ip**, BSTR **login**, BSTR **password**, BSTR **arch_password**, long **param**, long **port**) set up a connection to the Server.

- BSTR **ip** – IP address of the Video server;
- BSTR **login** – login to connect to the Server (can be blank);
- BSTR **password** – password to set up a connection to the Video server (can be blank);
- BSTR **arch_password** – password to access the archive (i.e. admin password, can be blank);
- long **param** – Server role:
  - 0 – video server;
  - 1 – backup archive;
  - 2 – videogate;
- long **port** – port to connect Video server. The parameter is mandatory.
  - if 0, 1 or 2 are passed, the connection is established with port 900, 901 or 902 correspondingly;
  - if 100 is passed, the connection is with port 10504;
  - if any onter value passed, the connection is with port number "port + 20000". For example, if port=900, the connection is established with server port 20900.

The connection to Server is set up **asynchronously**.

ShowCam

ShowCam(long **cam_id**, long **compress**, long **show**) shows/hides camera on the monitor.

- long **cam_id** – camera ID

- long **compress** – level of video compression 0-5 (for local camera =0). If set to -1, video stream is displayed directly from the camera without compression.
- long **show** – checkbox: show/hide camera (1/0)

DoReactMonitor

DoReactMonitor(BSTR **react_string**) – control over the monitor/cameras

- BSTR **react_string** – reaction string view

How to create react_string:

```
react_string =   “MONITOR||ARCH_FRAME_TIME|cam<3>,date<dd-mm-yy>,time<hh:mm:ss>”;
CamMonitor1->DoReactMonitor(react_string);
```

The result of calling the function with the parameter: camera 3 will be in the archive mode and the archive will be positioned to date «dd-mm-yy» and time «hh:mm:ss» (date and time are to be set in this format only).

The mode parameter takes the following values:

0 – video gate if it's specified (otherwise, video server).

1 – video server.

2 – long-time archive.

“MONITOR|**<id ignored>**|ARCH_FRAME_TIME|...”

Note:
Positioning accuracy can be specified in milliseconds, for instance:

```
DoReactMonitor(“MONITOR||ARCH_FRAME_TIME|cam<3>,date<02-10 05>,time<12:12:22.345>
```

RemoveAllCams

RemoveAllCams() : long – remove all cameras from the monitor

IsConnected

IsConnected() : boolean – method shows if the Video server is connected or disconnected.

GetCurIp

GetCurIp() : BSTR – returns the IP address of Server specified when calling **Connect**.

SendRawMessage

SendRawMessage(BSTR **msg**) – sends the command to be executed to Video server.

- BSTR **msg** – command string view

How to call a function:

```
m_Cam.SendRawMessage(“CAM|1|REC”);
```

```
m_Cam.SendRawMessage(“CAM|1|REC_STOP”);
```

```
m_Cam.SendRawMessage(“CAM|1|ARM”);
```

```
m_Cam.SendRawMessage(“CAM|1|DISARM”);
```

Disconnect

Disconnect() – disconnect from the Video server.

SetCallBackOptions

SetCallBackOptions(int **cam_id**, int **options**) – sets parameters of getting video from camera.

- int **cam_id** – camera ID (number).
- int **options** – options. The possible values of the **options parameter are:**
    - WithoutVideoFrame = 0x00 – do not send frames from the video module.
    - WithVideoFrame = 0x01 – send frames from the video module.
    - WithExtendedParams = 0x02 – get frames with extended parameters (time, fps, subtitles).
    - WithInformationLayout = 0x04 – display video in the window with control elements (context menu).
    - WithCompressedData = 0x08 – display video in the native format without decompression (if any).

- WithoutDecode = 0x10 – disable video decoding on the server.
- WithoutSubtitles=0x20 – disable subtitles.

Note. The options parameter is created the same as parameters of the CamMonitor.ocx component – see CamMonitor.ocx parameters.

CamMonitor.ocx events

OnCamListChange (long **cam_id**, long **action**) – occurs when there is connection with the Server or the number of cameras on the Server changes.

• long **cam_id** – camera ID.

• long **action** equals 1, if camera with **id** == **cam_id** exists, otherwise **action** == **0.**

This event occurs as many times as there are cameras on the Server. The negative value of the **cam_id** parameter (cam_id < 0) shows that **OnCamListChange** is not called.

If there are 3 cameras (1, 2, 3) on the Server, then the following events will occur one after another:

CamListChange(1,1)

CamListChange(2,1)

CamListChange(3,1)

CamListChange(-1,1)

Example:

Show the camera with cam_id =2 with **compress =1** compression level;

```
CamMonitor1CamListChange(long cam_id, long action)
{
   if(cam_id == -1)
   {
      CamMonitor1->ShowCam(2,1,1);
   }
}
```

Intellect HTTP API

General information on HTTP API

HTTP API is represented by web2 module (*Web-server 2.0*).

Note.
See Administrator’s Guide, Configuring the server for the clients connection via the Web-server 2.0 module section.

HTTP API allows the following features:

1. Get information about interactive maps: map list, map name, map layer list, layer parameters, layer background image, information about the list of points and an individual point on the layer (see Map).
2. Get information about object classes created on the Server, a list of states for the object class and information about status, icons for a specific state (see Object classes).
3. Get a list of objects created on the server, information about the individual object, the state of the object, the list of available actions with the object (see Objects).
4. Receive events from the Server both separately and by blocks (see Getting events).
5. Send commands to the server (see Sending commands to server).
6. Run macros (see Macros).
7. Work with video: get frames, request configuration, receive live and archive video, manage recording, arm and disarm cameras, manage telemetry (see Video).
8. Use notification systems to subscribe an application to APNS messages (see Notification).
9. Get live and archive sound (see Sound).

10. Send events and reactions to the core of the Intellect software (see Sending reactions and events to Intellect using HTTP request).

The following notation is used in the examples shown in this section:

- Port stands for port number.
- /web2 – web context where the web2 app operates. This is the web-app context.

Further the description will be omitted when query action is clear in the context.

Important!
URL, id of objects and file extension are case-sensitive.

Note.
Date and time are specified in RFC 3339 format, see details at http://www.ietf.org/rfc/rfc3339.txt

Getting events

Connection is not lost and events are always received.

action – type of event. Possible values: create, delete, update.

The fields below are optional:

objectId - id of object that is the source of event (always with update, delete, create).

state – id of a new object state (always with create. If the state has not changed, then there is no update state).

x, y – new coordinates (if changed).

Query:

http://example.com:[port]/web2/secure/feed/

Samples of response:

```
<message>
    <action>update</action>
    <objectId>CAM:1</objectId>
    <state>disconnected</state>
</message>


<message>
    <action>state</action>
    <objectId>CAM:1</objectId>
```

```
    <x>10.0</x>
    <y>123.9</y>
</message>

<message>
    <action>state</action>
    <objectId>CAM:1</objectId>
    <state>connected</state>
    <x>300.8</x>
    <y>670</y>
</message>

<message>
    <action>state</action>
    <objectId>CAM:1</objectId>
    <x>100</x>
    <y>100</y>
</message>

<message>
    <action>ping</action>
</message>
```


Getting events of video subsystem in blocks

http://example.com:[port]/web2/secure/events/

Parameters:

from – the oldest date of message search period. 2012-12-27T15%3A19%3A16.000%2B04%3A00

to – the latest date of message search period. 2012-12-27T15%3A19%3A16.000%2B04%3A00

count – maximum number of messages in reply [1, 200]. On default – 20. Server can return more messages if there are few messages in the database.

objectId – object id (CAM:1, GRAY:5 etc.). If parameter is not specified, events are returning for all.

Return codes:

200 - OK
400 - invalid parameter (e.g. date format)
500 - error
503 - error of core connection
504 - time-out (core failed to return data within 2000 milliseconds)

Examples of reply

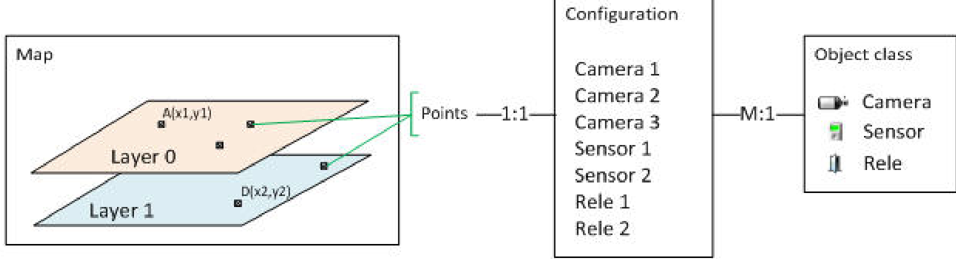
XML:


```
<events>
   <event>
      <description>Record is on</description>
      <id>{E56B09A0-1A50-E211-840E-005056C00008}</id>
      <objectId>CAM:1</objectId>
      <ts>2012-12-27T15:43:27+04:00</ts>
   </event>
   <event>
      <description>Record is off</description>
      <id>{4482F63F-1A50-E211-840E-005056C00008}</id>
      <objectId>CAM:1</objectId>
      <ts>2012-12-27T15:40:50+04:00</ts>
   </event>
   <event>
      <description>Record is off</description>
      <id>{35D4BE3E-1750-E211-840E-005056C00008}</id>
      <objectId>CAM:1</objectId>
      <ts>2012-12-27T15:19:16+04:00</ts>
   </event>
</events>
```

JSON:

[{
 "id" : "{E56B09A0-1A50-E211-840E-005056C00008}",
 "description" : "Record is off",
 "ts" : "2012-12-27T15:43:27.000+04:00",
 "objectId" : "CAM:1"
}, {
 "id" : "{4482F63F-1A50-E211-840E-005056C00008}",
 "description" : "Record is off",
 "ts" : "2012-12-27T15:40:50.000+04:00",
 "objectId" : "CAM:1"
}, {
 "id" : "{35D4BE3E-1750-E211-840E-005056C00008}",
 "description" : "Record is off",
 "ts" : "2012-12-27T15:19:16.000+04:00",
 "objectId" : "CAM:1"
}]

Maps



Several maps can be created on the server. Each map can consists of one or more layers. There are points on each layer. Each point corresponds to one of the objects in configuration.

Configuration – objects in Intellect. Each object represents the object of specific class. Each object has one state and the list of actions to be performed.

The object class describes its icons, possible states and possible actions with the object in each state.

Getting the list of maps

There can be 0 or more maps

http://example.com:[port]/web2/secure/kartas/

Sample of response:

```
<kartas>
<karta>
<id>plan</id>
<name>This is plan of a building</name>
</karta>
<karta>
<id>site</id>
<name>This is site around the building</name>
</karta>
</kartas>
```

Information on one map

plan – map id

http://example.com:[port]/web2/secure/kartas/**plan**/

Sample of response:

```
<karta>
    <id>plan</id>
    <name>This is plan of a building</name>
</karta>
```

The list of layers for specific map

plan – map id

There can be 1 or more layers

http://example.com:[port]/web2/secure/kartas/**plan**/layers/

Sample of response:

```
<layers>
```

```
    <layer>
        <height>1000</height>
        <id>base</id>
        <mapId>plan</mapId>
        <name>Base layer for plan</name>
        <width>1000</width>
        <zoomDef>1.0</zoomDef>
        <zoomMax>4.0</zoomMax>
        <zoomMin>0.25</zoomMin>
        <zoomStep>0.25</zoomStep>
    </layer>
</layers>
```

Parameters:

Height – height of layer's image in pixels;

Width – width of layer's image in pixels;

zoomMin – minimal zoom;

zoomMax – maximal zoom;

zoomStep – zoom step when zooming in or zooming out;

zoomDef – zoom by default.

Sample. If image width is 100 pixels, then width for 0.25 zoom is 100*0.25=25 pixels.

Information on a specific layer

Information on parameters is given in The list of layers for specific map section.

base – layer id

http://example.com:[port]/web2/secure/kartas/**plan**/layers/**base**/

Sample of response:

```
<layer>
    <height>1000</height>
```

```
    <id>base</id>
    <mapId>plan</mapId>
    <name>Base layer for plan</name>
    <width>1000</width>
    <zoomDef>1.0</zoomDef>
    <zoomMax>4.0</zoomMax>
    <zoomMin>0.25</zoomMin>
    <zoomStep>0.25</zoomStep>
</layer>
```

Layer background

http://localhost:8080/server-1.0/secure/kartas/**plan**/layers/**base**/image.[png|jpg]

Image in png or jpg format comes in response.

404 error comes to JPG, Jpg, JPEG, PNG query.

The list of point on the layer

http://example.com:[port]/web2/secure/kartas/**plan**/layers/**base**/points/

id is the same as id of object in configuration. Id is not necessarily equal to CAM:1. Id is to be considered as a line.

The coordinate plane is attached to the layer as follows:

I.e. x and y cannot be negative, but can be fractional.

Sample of response:


```
<points>
    <point>
        <id>CAM:1</id>
        <layerId>base</layerId>
        <mapId>plan</mapId>
        <x>100.0</x>
        <y>100.0</y>
    </point>
    <point>
        <id>CAM:2</id>
        <layerId>base</layerId>
        <mapId>plan</mapId>
        <x>200.0</x>
        <y>200.0</y>
    </point>
    <point>
        <id>GRAY:1</id>
        <layerId>base</layerId>
        <mapId>plan</mapId>
        <x>300.0</x>
        <y>300.0</y>
    </point>
    <point>
        <id>GRAY:2</id>
        <layerId>base</layerId>
        <mapId>plan</mapId>
        <x>400.0</x>
        <y>400.0</y>
```

```
    </point>
    <point>
        <id>GRELE:1</id>
        <layerId>base</layerId>
        <mapId>plan</mapId>
        <x>500.0</x>
        <y>500.0</y>
    </point>
    <point>
        <id>GRELE:2</id>
        <layerId>base</layerId>
        <mapId>plan</mapId>
        <x>600.0</x>
        <y>600.0</y>
    </point>
</points>
```


Information on a specific point on the layer

http://example.com:[port]/web2/secure/kartas/**plan**/layers/**base**/points/CAM:2 – request for information on a point corresponding to the camera with id 2.

Sample of response:


```
<point>
<id>CAM:2</id>
<layerId>base</layerId>
<mapId>plan</mapId>
<x>200.0</x>
<y>200.0</y>
</point>
```


Object classes

The list of object classes on the server

http://example.com:[port]/web2/secure/objectClasses

Sample of response:

<objectClasses>

<objectClass>

<id>GRELE</id>

</objectClass>

<objectClass>

<id>USERS</id>

</objectClass>

<objectClass>

<id>CAM</id>

</objectClass>

<objectClass>

<id>RIGHTS</id>

</objectClass>

<objectClass>

<id>GRAY</id>

</objectClass>

</objectClasses>

Specific object class

http://example.com:[port]/web2/secure/objectClasses/**GRELE**/

Sample of response:

<objectClass>

<id>GRELE</id>

</objectClass>

The list of states for a specific object class

http://example.com:[port]/web2/secure/objectClasses/**GRELE**/states/ - get the list of states for the **Relay** object class.

Sample of response:

```
<states>
   <state>
      <id>off</id>
   </state>
   <state>
      <id>on</id>
   </state>
   <state>
      <id>disabled</id>
   </state>
</states>
```

Information on a specific state

http://example.com:[port]/web2/secure/objectClasses/**[ObjectClass]**/states/**[State]**/

Sample:

http://example.com:[port]/web2/secure/objectClasses/**GRELE**/states/**off**/ - getting information on the OFF state for the **Relay** object class.

Sample of response:

```
<state>
   <id>off</id>
</state>
```

Getting the icon for a specific state

http://example.com:[port]/web2/secure/objectClasses/**[ObjectClass]**/states/**[State]**/image.png

Sample:

http://example.com:[port]/web2/secure/objectClasses/**GRELE**/states/**off**/image.png - getting the icon for the OFF state for the **Relay** object class.

The image in png format comes in response:

The list of events for a specific object class

GET

http://example.com:[port]/web2/secure/objectClasses/**GRELE**/events/ - getting the list of events for the **Relay** object class.

Sample of response:

XML


```
<events>
   <event>
      <id>23</id>
      <sid>grele.disable</sid>
      <description>Disable rele</description>
   </event>
   <event>
      <id>24</id>
      <sid>grele.enable</sid>
      <description>Enable rele</description>
   </event>
<baseObject>
   <CAM>
      <id>CAM:1</id>
      <name>Camera 1</name>
      <state>
         <id>disconnected</id>
      </state>
   </CAM>
   <GRELE>
      <id>GRELE:2</id>
      <name>Relay[GRELE] 2</name>
      <state>
         <id>off</id>
      </state>
```

```
    </GRELE>
    <GRELE>
        <id>GRELE:1</id>
        <name>Relay 1</name>
        <state>
            <id>disabled</id>
        </state>
    </GRELE>
    <GRAY>
        <id>GRAY:2</id>
        <name>Ray 2</name>
        <state>
            <id>alarmed</id>
        </state>
    </GRAY>
</baseObjects>
```

Objects

Getting list of objects

http://example.com:[port]/web2/secure/configuration/?pageItems=3&page=2 – returns the list of cameras added to the Web-server.

Parameters:

page – optional parameter. Sets the number of a page displayed as a result. By default: 1.

pageItems – optional parameter. Sets the number of objects displayed on the page. By default: 1000.

Important!
If there are many objects in the system (>1000) they are to be displayed by pages.

Processing of all objects is performed page by pare until an empty array is received.

JSON

```
[ {
  "type" : "CAM",
```

```
    "id" : "CAM:2",
    "extId" : "2",
    "name" : "Camera 2",
    "regionId" : "2.1",
    "state" : {
      "id" : "alarmed",
      "type" : "ALARM"
    },
    "presets" : [ ]
  }, {
    "type" : "CAM",
    "id" : "CAM:1",
    "extId" : "1",
    "name" : "Camera 1",
    "state" : {
      "id" : "armed",
      "type" : "NORMAL"
    },
    "presets" : [ ]
  }, {
    "type" : "GRAY",
    "id" : "GRAY:1",
    "extId" : "1",
    "name" : "Sensor 1",
    "state" : {
      "id" : "disconnected",
      "type" : "ALARM"
    }
  }, {
    "type" : "GRELE",
    "id" : "GRELE:2",
    "extId" : "2",
    "name" : "Relay 2",
    "state" : {
      "id" : "disabled",
      "type" : "NORMAL"
    }
  }, {
    "type" : "GRELE",
    "id" : "GRELE:1",
    "extId" : "1",
    "name" : "Relay 1",
    "regionId" : "2.1",
    "state" : {
      "id" : "disabled",
      "type" : "NORMAL"
    }
  }, {
```

```
  "type" : "GRAY",
  "id" : "GRAY:2",
  "extId" : "2",
  "name" : "Sensor 2",
  "state" : {
    "id" : "disconnected",
    "type" : "ALARM"
  }
} ]
```

Information on a specific object

http://example.com:[port]/web2/secure/configuration/GRAY:2/ - getting information on the **Sensor** object with id 2.

Sample of response:

```
<GRAY>
   <id>GRAY:2</id>
   <name>Ray 2</name>
   <state>
      <id>alarmed</id>
   </state>
</GRAY>
```

State of a specific object

http://example.com:[port]/web2/secure/configuration/GRAY:2/state/ - getting a state for the **Sensor** object with id 2.

Sample of response:

```
<state>
<fullState>ON,ARMED</fullState>
<id>armed</id>
<type>NORMAL</type>
</state>
```

The <fullState> contains a full object state as stored in the database, the <type> and <id> tags are object state in terms of HTTP API. Possible values of these parameters for a sensor are as follows:

Sensor state	Full list of states
Sensor disarmed	<fullState>DISARMED</fullState> <id>connected</id> <type>NORMAL</type>

Sensor disarmed + alarm	<fullState>ALARMED</fullState> <id>alarmed</id> <type>ALARM</type>
Sensor disarmed + alarm confirmed	<fullState>CONFIRMED</fullState> <id>confirmed</id> <type>NORMAL</type>
Sensor disarmed + on	<fullState>ON,DISARMED</fullState> <id>connected</id> <type>NORMAL</type>
Sensor disarmed + off	<fullState>DISARMED</fullState> <id>connected</id> <type>NORMAL</type>
Sensor disarmed + signal lost	<fullState>DISARMED,DETACHED_DISARM</fullState> <id>disconnected</id> <type>ALARM</type>
Sensor armed	<fullState>ARMED</fullState> <id>armed</id> <type>NORMAL</type>
Sensor armed + alarm	<fullState>ALARMED</fullState> <id>alarmed</id> <type>ALARM</type>
Sensor armed + alarm confirmed	<fullState>CONFIRMED</fullState> <id>confirmed</id> <type>NORMAL</type>
Sensor armed + on	<fullState>ON,ARMED</fullState> <id>armed</id> <type>NORMAL</type>
Sensor armed + off	<fullState>ARMED</fullState> <id>armed</id> <type>NORMAL</type>
Sensor armed + signal lost	<fullState>DETACHED_DISARM</fullState> <id>disconnected</id> <type>ALARM</type>

The list of available actions with the object in a specific state

The list of actions is requested not by the object class, but is taken in the context of a specific object as various user rights are possible for the objects of the same class.

Information on how to use this list is given in Sending commands to server section.

http://example.com:[port]/web2/secure/configuration/**GRAY:2**/state/actions/ - getting the list of available actions for the **Sensor** object with id 2.

Sample of response:

```
<actions>
    <action>
        <description>Disarm ray</description>
        <id>ray.disarm</id>
    </action>
    <action>
        <description>Confirm alarm</description>
        <id>ray.confirm</id>
    </action>
</actions>
```

If the object state considers no actions, then xml is:

```
<actions/>
```

Product version

http://example.com:[port]/web2/product/version

Such URL can be used to identify server

If the response is text/plain line like

Intellect 4.8.4

It means that the server supports the protocol described in this document. The line can change depending on the product version. This helps to distinguish 2 web servers with similar functionality but different protocols in different products.

Sending commands to server

PUT

http://example.com:[port]/web2/secure/configuration/**GRAY:2**/state/actions/**DISARM**/execute - sample of how to send the disarm **Sensor** with id 2 command to the sever.

Important!
The command name (in this example – DISARM) is to be in upper case.

Macros

In the section:

- Getting the list of macros
- Getting parameters of macros
- Macros execution on server request

Macros - predefined sequence of responses to certain events. Macros are created on the server and have IDs and names. They are similar to actions with objects, but are not attached to the object.

Getting the list of macros

GET

http://example.com:[port]/web2/secure/actions/

Sample of response:

```
<actions>
    <action>
        <description>Start recording by all cameras</description>
        <id>macro2</id>
    </action>
    <action>
        <description>Disarm all zones</description>
        <id>1</id>
    </action>
</actions>
```

Getting parameters of macros

Object has no extra parameters. The list of macros is sufficient.

GET

http://example.com:[port]/web2/secure/actions/**macro2**/ - getting parameters of macro with macro2 id.

Sample of response:

<action>

<description>Start recording by all cameras</description>

<id>macro2</id>

</action>

Macros execution on server request

PUT

http://example.com:[port]/web2/secure/actions/**macro2**/execute – request to execute macro with id macro2 on the server.

Video

Configuration request

GET

http://example.com:[port]/web2/secure/video/config.properties?version=4.7.8.0&login=XXX&password=YYY

Parameters:

- version – mandatory field. Client version (if protocol is changed). Now the **4.7.8.0** value is to be sent.
- login – optional field.
- password - optional field. It is in use if access is protected with the password.

Features of using

At the start it is not clear if the login and password are in use. To make it clear, send the following query:

GET

http://www.examplehost.com/web2/secure/video/config.properties?version=4.7.8.0

The server sends back the config.properties text file:

password.enabled=true

login.enabled=true

password.invalid=true#

Note.
symbol means the ending of configuration file.

When you get this file, it becomes clear that the password has been set and it is not correct. It is not correct because the login and password fields were blank.

Request the login and password and send configuration query to the server once again:

GET

http://www.examplehost.com/web2/secure/video/config.properties?version=4.7.8.0&login=XXX&password=YYY

If the password is correct or access is allowed without the password, then the server sends the following configuration:

password.enabled=true

login.enabled=true

password.invalid=false

cam.0.id=2

cam.0.name=Face

cam.0.rights=11

cam.1.id=3

cam.1.name=Camera 3

cam.1.rights=11

cam.2.id=5

cam.2.name=Camera 5

cam.2.rights=11

cam.2.telemetry_id=1.1

cam.count=3#

password.invalid=false means the specified password is not correct.

Note.
If access is allowed without the password, then password.enabled=false and configuration will be received on the first try.

cam.count=3 - total of cameras in the configuration (id starts at 0).

Get the data for each of 3 cameras in configuration.

cam.N.id – camera id.

cam.N.name - camera name.

cam.N.rights – rights.

cam.N.telemetry_id – telemetry id (there can be no value if there is no telemetry – then hide telemetry control elements).

cam.N.rights – rights (they are checked on the server; available on the client in order not to show the user odd options). The parameter is represented by flags. If the flag is set, then

the interface element is to be shown; if not – hidden.

static final int RIGHT_VIEW = 0x1; // live video is available(always 1)

static final int RIGHT_CONTROL = 0x2; // control (telemetry, arming and disarming)

static final int RIGHT_CONFIG = 0x4; // reserved

static final int RIGHT_HISTORY = 0x8; // access to archive

Video query

http://example.com:[port]/web2/secure/video/action.do?version=4.7.8.0&sessionid=FC126734&cam.id=5&login=XXX&password=YYY – request for video from a camera with ID 5.

cam.id – camera ID.

sessionid – any value.

If version 4.10.0.0 is specified in request, then the *stream obtained* is in *MJPEG* format without XML inserts – it can be displayed on a web page using the IMG tag in Chrome and FireFox browsers. This feature is implemented for both live and archive video.

Note.
Not more than 6 video streams can be received simultaneously.

Request sample:

http://10.0.36.158:8085/web2/secure/video/action.do?version=4.10.0.0&sessionid=1234567890&video_in=CAM:1&imageWidth=200&fps=1&login=1&password=1

Here is an example of using request results on the web page:

```
<html>
 <head/>
 <body>
  <img src="path to mjpeg"/>
 </body>
</html>
```

Format of main stream

Stream of following view will be received in reply

```
HTTP/1.0 200 OK

Connection: close

Server: ITV-Intellect-Webserver/4.9.0.0

Cache-Control: no-store,no-cache,must-revalidate,max-age=0
```

```
Pragma: no-cache
Date: Mon, 13 Jan 2013 10:44:27 GMT
Content-Type: multipart/mixed;boundary=videoframe

--videoframe
Content-Type: text/xml
Content-Length: 138

<video_in>
  <sessionid>FC126734</sessionid>
  <video_in>CAM:5</video_in>
  <newstate>started</newstate>
  <errcode>100</errcode>
</video_in>
 --videoframe
Content-Type: image/jpeg
Content-Length: 23978
X-Width: 320
X-Height: 240
X-Time: 2013-03-15T10:51:44.314+04:00
X-Timestamp: 0.000

 <jpeg image>
--videoframe
Content-Type: image/jpeg
Content-Length: 23651
X-Width: 320
X-Height: 240
X-Time: 2013-03-15T10:51:44.314+04:00
```

```
X-Timestamp: 0.152


<jpeg image>
```

Where:

- X-Width - image width.
- X-Height- image height.
- X-Time - absolute time of frame forming.
- X-Timestamp - relative frame time in seconds (relatively stream head).

In case of stream end due to the fault of server, the end packet can be received:

```
--videoframe
Content-Type: text/xml
Content-Length: 106
<video_in>
  <sessionid>FC126734</sessionid>
  <video_in>CAM:5</video_in>
  <newstate>closed</newstate>
  <errcode>103</errcode>
</video_in>
```

- sessionid - session ID (the same as at start).
- video_in - camera ID.
- errcode - error code:
  - 100 – error absence.
  - 101 – too many connected users.
  - 102 – invalid password (theoretically, it's possible to change password at any moment).
  - 103 – unavailable video.
  - 104 – old client version. Update version.

Managing records

Start recording

GET

http://example.com:[port]/web2/secure/video/action.do?version=4.7.8.0&sessionid=29101F1&cam.id=1&target=CAM&targetid=1&command=REC& login=XXX&password=YYY

Stop recording

GET

http://example.com:[port]/web2/secure/video/action.do?version=4.7.8.0&sessionid=29101F1&cam.id=5&target=CAM&targetid=1&command=REC_STOP&login=XXX&password=YYY

Here targetid==cam.id

Arming and disarming the camera

Arming

GET

http://example.com:[port]/web2/secure/video/action.do?version=4.7.8.0&sessionid=29101F1&cam.id=1&target=CAM&targetid=1&command=ARM& login=XXX&password=YYY

Disarming

GET

http://example.com:[port]/web2/secure/video/action.do?version=4.7.8.0&sessionid=29101F1&cam.id=5&target=CAM&targetid=1&command=DISARM&login=XXX&password=YYY

targetid==cam.id

PTZ control

GET

http://example.com:[port]/web2/secure/video/action.do?version=4.7.8.0&sessionid=29101F1&cam.id=5&target=PTZ&targetid=1.1&command=RIGHT&login=XXX&password=YYY&speed=2

Parameters:

command – required parameter. It takes the following values:

- RIGHT
- UP
- LEFT
- DOWN
- ZOOM_IN
- ZOOM_OUT
- GO_PRESET – go to the specified preset.
- POINTMOVE – zooming of selected point on the image (x,y).
- AREAZOOM – zooming of selected area on the image (x,y,w,h).

speed – required parameter. Command-processing speed (from 0 to 10). It is recommended to use low values due to delays while controlling by network

cam.id – required parameter. Camera ID.

target – required parameter. Always is equal to PTZ.

targetid – required parameter. Id of PTZ connected with camera (is sent in the SETUP configuration).

preset – number of preset. Required parameter only for the command=GO_PRESET. Otherwise its value is ignored.

x – x coordinate relatively the image size. It takes values from 0.0 to 1.0. Required parameter only for the command=POINTMOVE or command=AREAZOOM. Otherwise its value is ignored.

y – y coordinate relatively the image size. It takes values from 0.0 to 1.0. Required parameter only for the command=POINTMOVE or command=AREAZOOM. Otherwise its value is ignored.

w – width of zooming area relatively the image size. It takes values from 0.0 to 1.0. Required parameter only for the command=AREAZOOM. Otherwise its value is ignored.

h – height of zooming area relatively the image size. It takes values from 0.0 to 1.0. Required parameter only for the command=AREAZOOM. Otherwise its value is ignored.

Thumbnails request

Method 1

http://example.com:[port]/web2/secure/video/image.jpg?cam.id=1&version=4.7.8.0

Parameters:
cam.id – required. Camera ID.
width – optional. Value can be in [64, 1600] range. Server automatically rounds width to larger value divisible by 4.
height – optional. Value can be in [30, 1200] range.
version – optional. client version (for protocol exchange). It's required to send "4.7.8.0" value now.
login – optional. Login;
password – optional. If password access is set.

Size of returned image is taken from video stream if width and height parameters are not set.

Returned value:
jpeg image of approximately requested size.

In case of error the http error code is returned:
404 – camera disabled or not in use (disabled);
403 – invalid password;
426 – old client version;
429 – too many requests;
444 – camera signal is lost or camera disabled (coaxial conductor disconnected from card);
503 – archive error.

Example:
Get image from camera 5, width is approximately 85 pixels:
http://localhost:8079/web2/secure/video/image.jpg?cam.id=5&width=85&version=4.7.8.0&login=username&password=secret

Jpg image of width 88 pixels or error code and zero length body (i.e. only headings) will be received in reply).

Method 2

http://example.com:[port]/web2/secure/video/action.do?version=4.9.0.0&command=frame.video&video_in=CAM:1&imageWidth=400&imageHeight=200

cam.id – camera ID.

sessionid – any value.

imageWidth – requested frame width.

imageHeight – requested frame height.

Example:

Receive a frame from Camera 1 with a height of 400 pixels.

http://192.168.0.79:8085/web2/secure/video/action.do?version=4.9.0.0&command=frame.video&video_in=CAM:1&imageWidth=400

Using the archive

In the section:
• Getting list of records - "arc.intervals" • Getting one frame from archive - "arc.frame" • Getting video from archive - "arc.play" • Getting list of records (the second way)

Video archive stream is sent on the same format as live video.

Getting list of records - "arc.intervals"

GET

http://example.com:[port]/web2/secure/video/action.do?version=4.9.0.0&sessionid=29101F1&video_in=CAM:5&command=arc.intervals&time_from=2013-03-20T00:00:00.000+04:00&time_to=2013-03-22T23:59:59.999+04:00&max_count=100&split_threshold=10399&login=XXX&password=YYY

Required parameters:

command=arc.intervals – command to receive list of records

video_in – camera ID.

time_from – start of interested time range.

Optional parameters:

time_to – start and end of interested time range.

max_count – maximal number of records in reply

split_threshold – time for combining several intervals (in seconds). Intervals, distance between which will be less than specified value, will be combined in one.

In return **XML** will be received**:**

```xml
<?xml version="1.0" encoding="UTF-8"?>
<records>
    <record>
        <from>2011-09-01T00:00:00-05:00</from>
        <to>2011-09-01T00:00:35-05:00</to>
    </record>
    <record>
        <from>2011-09-01T00:00:35-05:00</from>
        <to>2011-09-01T00:01:10-05:00</to>
    </record>
</records>
```

Getting one frame from archive - "arc.frame"

GET

http://example.com:[port]/web2/secure/video/action.do?version=4.9.0.0&sessionid=29101F1&video_in=CAM:5&command=arc.frame&time=2013-03-22T13:04:52.312+04:00&range=0.1&login=XXX&password=YYY

Required parameters:

command=arc.frame - command for one frame;

video_in - camera ID;

time - interested time.

Optional parameters:

range - time in seconds to specify search range of the nearest frame relatively the time parameter (the nearest frame all over archive is searched if this parameter is not specified);

imageWidth - width in pixels (is counted automatically with saving proportions if it isn't specified or equal 0);

imageHeight - height in pixels (is counted automatically with saving proportions if it isn't specified or equal 0);

fps - maximal frame frequency per second (if it isn't specified or equal 0, frame frequency won't be limited if).

In return http-headings and the nearest frame from the [time - range, time + range] range in the jpeg format will be received. The reply body will be empty if there is no frame in the range.

Getting video from archive - "arc.play"

GET

http://example.com:[port]/web2/secure/video/action.do?version=4.9.0.0&sessionid=29101F1&video_in=CAM:5&command=arc.play&time_from=2013-03-22T13:04:52.312+04:00&time_to=2013-03-22T13:16:31.873+04:00&login=XXX&password=YYY

Required parameters:

command=arc.play - command to get video from archive;

video_in - camera ID;

time_from - start time of archive playing back.

Optional parameters:

time_to - completion time of archive playing back (if parameter is not specified, all archive will play back);

imageWidth - width in pixels (is counted automatically with saving proportions if it isn't specified or equal 0);

imageHeight - height in pixels (is counted automatically with saving proportions if it isn't specified or equal 0);

maximal frame frequency per second (if it isn't specified or equal 0, frame frequency won't be limited if).

End packet with newstate=closed and errcode=100 will be received when stream completion.

Getting list of records (the second way)

GET

http://example.com:[port]/web2/secure/archive/**CAM:2/[2011-12-30|2011-12]?[splitThreshold=50]&[days=1]**

splitThreshold – if difference between end of previous record and start of next record less than specified value (in milliseconds), than records will be combined in one. Specify splitTreshold=0. [default: 50] not to combine records.

days - number of days from the current, for which archive is required. [default: 1]

All time is interpreted as local time for server.

Get records for 18 November 2013 and combine all records, interval between which less than 2000 milliseconds:

http://example.com:[port]/web2/secure/archive/**CAM:1/2013-10-18/?splitTreshold=2000**

Get records for 10 days from 18 November 2013:

http://example.com:[port]/web2/secure/archive/**CAM:1/2013-10-18/?days=10**

XML:

```
<?xml version="1.0" encoding="UTF-16"?>
<days>
    <day>
        <id>2013-11-10T00:00:00-02:00</id>
        <records>
            <from>2013-11-10T18:44:01.579-02:00</from>
            <to>2013-11-10T18:44:09.717-02:00</to>
        </records>
    </day>
    <day>
        <id>2013-11-18T00:00:00-02:00</id>
        <records>
            <from>2013-11-18T18:38:30.252-02:00</from>
            <to>2013-11-18T18:38:56.942-02:00</to>
        </records>
        <records>
            <from>2013-11-18T18:39:08.321-02:00</from>
            <to>2013-11-18T18:39:10.080-02:00</to>
        </records>
    </day>
</days>
```

JSON:

```
[ {
  "id" : "2013-11-10T00:00:00.000-02:00",
  "records" : [ {
    "from" : "2013-11-10T18:44:01.579-02:00",
    "to" : "2013-11-10T18:44:09.717-02:00"
  } ]
}, {
  "id" : "2013-11-18T00:00:00.000-02:00",
  "records" : [ {
    "from" : "2013-11-18T18:38:30.252-02:00",
    "to" : "2013-11-18T18:38:56.942-02:00"
  }, {
    "from" : "2013-11-18T18:39:08.321-02:00",
    "to" : "2013-11-18T18:39:10.080-02:00"
  } ]
} ]
```


Getting records for a month (shows days of September at which there are records):

http://example.com:[port]/web2/secure/archive/**CAM:2/2011-12/**

XML:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<days>
    <day>
        <id>2011-09-02T00:00:00-05:00</id>
    </day>
    <day>
        <id>2011-09-03T00:00:00-05:00</id>
    </day>
    <day>
        <id>2011-09-05T00:00:00-05:00</id>
    </day>
</days>
```

JSON:

```json
[ {
  "id" : "2011-09-01T00:00:00-0500",
  "records" : [ ]
}, {
  "id" : "2011-09-03T00:00:00-0500",
  "records" : [ ]
}, {
  "id" : "2011-09-01T00:00:00-0500",
  "records" : [ ]
} ]
```

If there are no records, the following will be received

XML:

<days/>

JSON:

[]

Notification

In the section:
• Subscribing • Subscription cancellation • APN message format

APNS(iOS), C2DN (Android), etc. notification systems are in use.

deviceid – device token (APNs), registration id (C2DN), etc.;

username – login. The field can be blank.

Subscribing

When the app connects to service, the APNS messages subscription is performed. So when the app is closed, device will receive events notifications.

POST

http://example.com:[port]/web2/secure/subscription/

Reply with “201 Created” code means that subscribing is performed successfully.

Code 400 means that specified parameters are invalid (deviceId is not to be empty, it's length should be from 5 to 150 symbols and contain only digits and letter of English alphabet).

The POST body should contain information about created subscribing. Only JSON format is received. It's required to specify the Content-Type title properly.

Sample of response:

JSON

Content-Type : application/json

{

"username" : "johndoe",

"deviceid" : "somedeviceid"

}

Subscription cancellation

Subscription is cancelled in the following cases:

- The user subscribed to events from other device;
- Device token or registration id is changed;
- Another user subscribed to events from this device;
- Subscription is cancelled manually.

DELETE

http://example.com:[port]/web2/secure/subscription/[deviceId]

Reply with “204 No Content” code means that subscribing is performed successfully.

APN message format

```
{
"aps" : {
        "alert" : "Motion Detected",
        "badge" : 2 //ordinal number of the message. Numbers are given one after another after the latest subscribing.
     },
  "e" : {
       "srv" : "XXX",  //server id. Unique in one iOS device
       "stt" : 88,   //state id(see The list of states for a specific object class)
       "obj" : "6", //object id
       "ts" : "2010-08-02T23:30:00Z" //time of sending the event
     }
}
```

Sound

Getting live sound

GET

http://example.com:[port]/web2/secure/video/action.do?version=4.9.0.0&sessionid=FC126734&command=audio.play&audio_in=MIC:5&format=L16&login=XXX&password=YYY

sessionid – session ID (is not in use yet).
audio_in – audio stream ID.
format – format of audio data (only L16 yet).

Audio packets of the following view will be received:

```
HTTP/1.0 200 OK
Connection: close
Server: ITV-Intellect-Webserver/4.9.0.0
Cache-Control: no-store,no-cache,must-revalidate,max-age=0
Pragma: no-cache
Date: Mon, 13 Jan 2013 10:44:27 GMT
Content-Type: multipart/mixed;boundary=audioframe

--audioframe
Content-Type: text/xml
Content-Length: 138

<audio_in>
  <sessionid>FC126734</sessionid>
  <audio_in>MIC:5</audio_in>
  <newstate>started</newstate>
  <errcode>100</errcode>
</audio_in>
--audioframe
Content-Type: audio/L16;rate=8000;channels=1
Content-Length: 1024
X-Time: 2013-03-22T13:16:31.371+04:00

<audio packet PCM16>
--audioframe
Content-Type: audio/L16;rate=8000;channels=1
Content-Length: 1278
X-Time: 2013-03-22T13:16:31.873+04:00
```

<audio packet PCM16>

To stop stream without connection break send the following command in the same connection

GET
http://example.com:[port]/web2/secure/video/action.do?version=4.9.0.0&sessionid=29101F1&command=audio.stop&audio_in=MIC:5&login=XXX&password=YYY

The end xml packet will be received::

```
--audioframe
Content-Type: text/xml
Content-Length: 106
<audio_in>
  <sessionid>FC126734</sessionid>
  <audio_in>MIC:5</audio_in>
  <newstate>closed</newstate>
  <errcode>100</errcode>
 </audio_in>
```

Playing sound from archive

GET

http://example.com:[port]/web2/secure/video/action.do?version=4.9.0.0&sessionid=29101F1&command=arc.play&audio_in=MIC:5&format=L16&time_from=2013-03-22T13:16:31.873+04:00&time_to=2013-03-22T13:04:52.312+04:00&login=XXX&password=YYY

audio_in – audio stream ID;

format – requested format (only L16 yet);

time_from - start time of archive playing back;

time_to - end time of archive playing back.

The stream will be received in the same view as in case of live sound.

The end xml packet will be received when data completion (as when getting live sound – see Getting live sound).

Sending live sound

Sending of sound is performed by serial communication of packets using commands:

POST

http://example.com:[port]/web2/secure/video/action.do?version=4.9.0.0&sessionid=FC126734&command=audio.receive&audio_out=SPEAKER:3&login=XXX&password=YYY

Content-type: audio/L16;rate=8000;channels=1
Connection: keep-alive

Then audio packet transmission will perform.

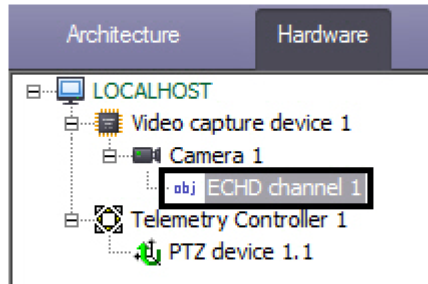
Format of sound – only L16.
rate – any rational value.
channels – from 1 to 6.

Commands used for ECHD integration

ECHD means 'Unified data center' state information system.

The description of http requests that are used for integration of *Intellect* with ECHD is given in this section. For requests operation they are to be enabled at the stage of system configuration – see Enabling the processing of ECHD requests and selecting rtsp server.

Moreover, requests are performed only for cameras under which the **ECHD channel object is created**.



Archive downloading

GET

http://example.com:[port]//downloadarchivefile?camera id =1&fromdatetime=2014-10-01T00:00:00&todatetime=2014-10-01T01:20:05

Example:

GET http://192.168.15.182:8095/downloadarchivefile?camera id =1&fromdatetime=2014-10-01T00:00:00&todatetime=2014-10-01T01:20:05

Response

HTTP/l.1 200 OK

Content-Type: application/octet-stream

Archive export

In the section:
• Creating archive export request • Getting export status • Deleting archive

By default, the archive is exported in mp4 format. Change the format using the ExportContainerFormat registry key (see Registry keys reference guide). The export of the archive in H.264 or MPEG4 formats is supported.

Creating archive export request

Example:

POST http://192.168.15.182:8096/createarchivetask
Content Type: application/json
Content:

{

"CameraId": "1",

"From": "2016-06-27T15:10:00.00Z",

"To": "2016-06-27T15:20:00.00Z"

}

Response:

{

"CameraId" : "1",

"From" : "2016-06-27T15:10:00.00Z",

"To" : "2016-06-27T15:20:00.00Z",

"ArchiveTaskId" : "084b56a5-bd49-4327-82db-9bc911f7ff96",

"ErrorMessage" : null,

"State" : "Created"

}

The folder with the corresponding name (084b56a5-bd49-4327-82db-9bc911f7ff96) and mp4 file inside it is created in the export folder (by default C:\Users\User\Documents\Intellect\export).

Getting export status

Example:
GET http://192.168.15.182:8096/getarchivetaskstatus?archivetaskid=104b38d4-07d7-4d2f-84da-49b3e255d2bf

Response:

{

"Percents" : 100,

"Url" : "http://192.168.15.182:8096/download?file=104b38d4-07d7-4d2f-84da-49b3e255d2bf",

"CameraId" : "1",

"From" : "2016-06-27T15:10:00.000+03:00",

"To" : "2016-06-27T15:11:00.000+03:00",

"ArchiveTaskId" : "104b38d4-07d7-4d2f-84da-49b3e255d2bf",

"ErrorMessage" : "null",

"State" : "ReadyForDownload"

}

The required mp4 file can be downloaded using the link in the URL bar.

Deleting archive

Example:
DELETE http://192.168.15.182:8096/removearchive?archivetaskid=084b56a5-bd49-4327-82db-9bc911f7ff96

Response:

{

"ArchiveTaskId" : "084b56a5-bd49-4327-82db-9bc911f7ff96",

"ErrorMessage" : null,

"Success" : true

}

The corresponding folder with mp4 file will be deleted from the export folder.

List of cameras and their parameters

In the section:

- GetCameras
- GetDeviceInfo

GetCameras

GET

http://example.com:[port]/getcameras

Returns the list of IDs of cameras registered on the video encoder/video server. Additionally, it can contain information on the status of the video surveillance device.

Example:

GET http://192.168.15.182:8095/getcameras

Response:

{

"cameras": [

{

"id": 1,

"channe1": 1,

"status": "working"

},

{

"id": 2,

channel ": 2,

"status": "signallost"

}

]

}

GetDeviceInfo

GET http://example.com:[port]/getdeviceinfo

Returns information about the device (firmware version, manufacturer, model and serial number).

Example:

GET http://192.168.15.182:8095/getdeviceinfo

Response

{

"firmware version": "1.2.3 Rev B.",

"vendor": "Vendor Title Ltd"

"model": "Device Model",

"serial_number": "12345ABCDEF",

"ptz-status": "not supported"

}

Ranges of available archive recordings

GET

http://example.com:[port]/getarchiveranges?cameraid=1

Returns time periods over which archive recordings from the specified video surveillance device are available.

Note.
By default fragments are merged if the interval between them is less than 5 seconds. This time period can be changed using the SplitArchiveIntervals registry key (see Registry keys reference guide).

Example:

GET http://192.168.15.182:8095/getarchiveranges?cameraid=1

Response**:**

{

"cameraid": 1,

"ranges": [

{

"from": 1412121600, //unixtime

"to": 1412172000

},

{

"from": 1412186400,

"to": 1412188200

}]

}

Video surveillance device features management

In the section:
- Discrete motion   - Relative motion   - Setting position of video surveillance device   - Getting position of video surveillance device   - Focusing   - Iris control   - Night mode   - Backlight   - BW mode

Discrete motion

https://example.com:[port]/?cameraID={1}& ip={2}& loqin={3}&pass={4}& action={5}&x={6}&y={7}&z ={8}&modelName={9}

Atomic shift of video surveillance device in the specified direction.

cameraID - ID of video surveillance device.

ip - IP address of video surveillance device.

login - account of video surveillance device.

pass - access password to video surveillance device.

action - command name *[degreesmove].*

x - PAN rotation [-180 ..0.. 180].

y - TILt rotation [-180 ..0.. 180].

z - zoom in/out [0.. 100].

modelName – model of video surveillance device.

Relative motion

https://example.com:[port]/?cameraID={1}&ip={2}&loqin={3}&pass={4}&action={5}&x={6}&y={7}&z={8}&modelName={9}

Rotation of video surveillance device compared with current position. Viewing area of video surveillance device is divided by a grid where central point's coordinates are (x:0, y:0), top left (x:-7, y:7), bottom right (x:7, y:-7). Video surveillance device must be rotated in a way that object appears in the center of the grid.

'Optical' error caused by the distance to the visible object is allowed.

Error caused by sphere-to-plane projection must be compensated.

cameraID - ID of video surveillance device.

ip - IP address of video surveillance device.

login - account of video surveillance device.

pass - access password to video surveillance device.

action - command name *(degreestove2).*

x - PAN rotation [-7..0..7].

y - tILt rotation [-7..0..7].

Z – zoom in/out [-1..0.. 1].

modelName - model of video surveillance device.

Setting position of video surveillance device

https://example.com:[port]/?cameraID={1}&ip={2}&login={3}&pass={4}&action={5}&x={6}&y={7}&z={8}&modelName={9}

Setting position of video surveillance device in degrees compared with ‘0’ position.

cameraID - ID of video surveillance device.

ip - IP address of video surveillance device.

login - account of video surveillance device.

pass - access password to video surveillance device.

action - command name *(setposition).*

x - PAN setting [-180 ..0.. 180].

y - TILt setting [-180 ..0..180].

Z – zoom value [0.. 100].

modelName - model of video surveillance device.

Getting position of video surveillance device

https://example.com:[port]/?cameraID={1}&ip={2}&login={3}&pass={4}&action={5}&modelName={6}

Getting position of video surveillance device in pAN and TILt in degrees as well as current zoom values.

Response in JSON format:

{"y":56, "x":105, "z":0}

cameraID - ID of video surveillance device.

ip - IP address of video surveillance device.

login - account of video surveillance device.

pass - access password to video surveillance device.

action - command name *(getposition).*

modelNate - model of video surveillance device.

Focusing

https://example.com:[port]/?cameraID={1}&ip={2}&login={3}&pass={4}&action={5}&x={6}&y={7}&z={8}&modelName={9}

Video surveillance device’s focus control command where z parameter controls over focus:

1: Focus in

-1 : Focus out

0: Auto

cameraID - ID of video surveillance device.

ip - IP address of video surveillance device.

login - account of video surveillance device.

pass - access password to video surveillance device.

action - command name *(focus).*

x - not in use [0].

y - not in use [0].

z – focus control [-1..0..1].

modelNate - model of video surveillance device.

Iris control

https://example.com:[port]/?cameraID={1}&ip={2}&login={3}&pass={4}&action={5}&x={6}&y={7}&z={8}&modelName={9}

Video surveillance device's iris control command where z parameter controls over iris:

1: Open iris

-1: Close iris

0: Auto

cameraID - ID of video surveillance device.

ip - IP address of video surveillance device.

login - account of video surveillance device.

pass - access password to video surveillance device.

action - command name *(iris).*

x - not in use [0].

y - not in use [0].

z – iris control [-1..0..1].

modelNate - model of video surveillance device.

Night mode

https://example.com:[port]/?cameraID={1}&ip={2}&login={3}&pass={4}&action={5}&x={6}&y={7}&z={8}&modelName={9}

The following operation modes of video surveillance device are available:

1: Day mode

-1 : Night mode

cameraID – ID of video surveillance device.

ip – IP address of video surveillance device.

login - account of video surveillance device.

pass – access password to video surveillance device.

action – command name *(switch_day_night).*

x – not in use [0].

y - not in use [0].

z – mode control [-1..0..1].

modelNate - model of video surveillance device.

Backlight

https://example.com:[port]/?cameraID={1}&ip={2}&login={3}&pass={4}&action={5}&x={6}&y={7}&z={8}&modelName={9}

The following backlight operation modes are available:

1: Enable

-1: Disable

cameraID - ID of video surveillance device.

ip - IP address of video surveillance device.

login - account of video surveillance device.

pass - access password to video surveillance device.

action - command name *(backlight).*

x - not in use [0].

y - not in use [0].

z - mode control[-1..0..1].

modelNate - model of video surveillance device.

BW mode

https://example.com:[port]/?cameraID={1}&ip={2}&login={3}&pass={4}&action={5}&x={6}&y={7}&z={8}&modelName={9}

The following operation modes of video surveillance device are available:

1: Enable

-1: Disable

cameraID - ID of video surveillance device.

ip - IP address of video surveillance device.

login - account of video surveillance device.

pass - access password to video surveillance device.

action - command name *(switch_color).*

x - not in use [0].

y - not in use [0].

z - mode control[-1..0..1].

modelName - model of video surveillance device.

Working with video streams

In the section:
• GetLiveUrl(CameraId) • GetArclliveURL(CameraId, FromDatetime, toDatetime)

GetLiveUrl(CameraId)

GET

http://example.com:[port]/getliveurl?cameraid=1

Returns rtsp URL of “live” video stream for the specified camera.

Example:

GET http://192.168.15.182:8095/getliveurl?cameraid=1

Response

{

"rtspurl": "rtsp://device-address/somelivemediastream0"

}

GetArclliveURL(CameraId, FromDatetime, toDatetime)

GET

http://example.com:[port]/getarchiveurl?cameraid=1&fromdatetime=2014-10-01T00:00:00&todatetime=2014-10-01t01:20:05

Returns rtsp URL of archive video stream received from video encoder for the specified camera starting from FromDateTime (and, optionally, ending at endDatetime).

Example:

GEt http://192.168.15.182:8095/getarchiveurl?cameraid=1&fromdatetime=2014-10-01T00:00:00&todatetime=2014-10-01t01:20:05

Response

{

"rtspurl": "rtsp://deviceladdress/somearchivemediastream?somedatetimetoken"

}

Sending reactions and events to Intellect using HTTP request

Intellect gets commands and events to 10112 port:

http://[Intellect Server IP]:10112/intellect_core/React?command="[Intellect format command]"

http://[Intellect Server IP]:10112/intellect_core/Event?command="[Intellect format command]"

Note.
The specified port can be changed using the RestPort registry key – see details in Registry keys reference guide.

Examples:

Add captions to video from camera 2 via HTTP request:

http://localhost:10112/intellect_core/React?command="CAM|2|ADD_SUBTITLES|command<Some text\n!>"

Generate alarm on camera 2 via HTTP request:

http://localhost:10112/intellect_core/Event?command="CAM|2|MD_START"

When getting such commands, standard events and reactions will be generated in *Intellect* – they can be used in scripts and macros (see Creating and using macros section of Administrator's Guide as well as Programming Guide (JScript)).

Sending HTTP API commands using curl tool

To test HTTP API one can send HTTP API commands using curl tool. This tool is available at https://curl.haxx.se/.

To use the tool start the Windows command line and go to <curl installation directory>\curl-7.46.0-win64\bin folder.

Find the example of the command to create the archive export task below (see Archive export):

curl -H "Content-Type: application/json" -X POST -d "{ \"CameraId\": \"1\", \"From\": \"2017-12-26T10:58:00.00Z\", \"To\": \"2017-12-26T11:00:00.00Z\" } " http://127.0.0.1:8095/createarchivetask

Response example:

{ "CameraId" : "1", "From" : "2017-12-26T10:58:00.00Z", "To" : "2017-12-26T11:00:00.00Z", "ArchiveTaskId" : "084b56a5-bd49-4327-82db-9bc911f7ff96", "ErrorMessage" : null, "State" : "Created" }

Intellect software Integration Guide. Postscript

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

1. Administrator’s Guide;
2. Operator’s Guide;
3. Installing and configuring security system components guide;
4. Programming Guide (JScript).

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. DDI file structure

The table below contains a description of the fields of the table from the **Names** tab (the **<Objects>** section):

Field	**Description**
Name (<ObjectName>)	Object ID
Visible name (<VisibleName>)	Visible name
Group name (<GroupName >)	The name of a group of objects. Used for grouping objects in Intellect's settings tree

The table below contains a description of the fields of the table from the **Events** tab (the **<Events>** section):

Field	**Description**
Name (<EventName>)	Event ID
Description (<EventDescription>)	Event description recorded in the event log
Event handling (<EventType>)	Used to set the background color in the event log normal – no background color; alarm – red window; information – blue window
Sound support (<IsSoundEnabled>)	A sound file is played when a message arrives.
Do not send over network (<IsNetworkDisabled>)	Messages will not be sent over the network
Do not log (<IsProtocolDisabled>)	Events will not be recorded in the event log

Windows log (<IsWindowsLogEnabled>)	Record messages in the Windows log. *Note: Recording in the Windows log is impossible for an event, if* *the event is not logged*

The table below contains a description of the fields of the table from the **Reacts** tab (the **<Reacts>** section):

Field	**Description**
Name (<ReactName>)	Reaction name
Description (<ReactDescription>)	The reaction description shown in the context menu after a right-click on the object icon on the *Map*
Flags (<IsReactArm>)	Reaction scope flags: perform actions for a single object or for a group of objects from the same section

The table below contains a description of the fields of the table from the **Icons** tab (the **<Icons>** section):

Field	**Description**
Filename (<FileName>)	A BMP file name (the part that serves as an image ID). An image ID allows you to use multiple BMP files to show objects of the same type on the *Map*. (see Section Using the ddi.exe Tool to Work with DDI files)
Name (<IconName>)	A description of the BMP file of an object

The table below contains a description of the fields of the table from the **States** tab (the **<States>** section):

Field	**Description**
Name (<StateName>)	State name
Image (<ImgName>)	BMP file name (the part that serves as a state ID). (see Section Using the ddi.exe Tool to Work with DDI files). Note: The *Map* may show objects using lines (that is, without using BMP files). In this case, when an object changes its state, the line color changes. *For a state, the color (RGB) is set as follows:* **<State>$R:G:B**
Description (<StateDescription>)	State description
Flashing (<IsStateFlashing>)	Display on the *Map*: normal – the icon does not flash , alarm – the icon flashes

The table below contains a description of the fields of the table from the **Transition Rules** tab (the **<Rules>** section):

Field	**Description**
Event (<EventName>)	The event that triggers the transition
Transition From (<FromStateName>)	The start state (that is, the state that we transition from)

Transition To (<ToStateName>)	The end state (that is, the state that we transition to)

APPENDIX 2. NissObjectDLLExt and CoreInterface class declarations

On the page:
• CoreInterface • NissObjectDLLExt

CoreInterface

```
class CoreInterface

{

public:

        virtual BOOL DoReact    (React&) = 0;

        virtual BOOL NotifyEvent(Event&) = 0;

        virtual void SetupACDevice(LPCTSTR objtype, LPCTSTR objid, LPCTSTR objtype_reader) = 0;



        virtual   BOOL    IsObjectExist(LPCTSTR objtype, LPCTSTR id) = 0;

        virtual   BOOL    IsObjectDisabled(LPCTSTR objtype, LPCTSTR id) = 0;

        virtual Msg FindPersonInfoByCard(LPCTSTR facility_code, LPCTSTR card) = 0;

        virtual Msg FindPersonInfoByExtID(LPCTSTR external_id) = 0;

        virtual   CString GetObjectName (LPCTSTR objtype, LPCTSTR id) = 0;

        virtual   CString GetObjectState(LPCTSTR objtype, LPCTSTR id) = 0;
```

```
virtual   void       SetObjectState(LPCTSTR objtype, LPCTSTR id, LPCTSTR state) = 0;
virtual   BOOL    IsObjectState(LPCTSTR objtype, LPCTSTR id, CString state) = 0;


virtual   CString GetObjectParam    (LPCTSTR objtype, LPCTSTR id, LPCTSTR param) = 0;
virtual   int  GetObjectParamInt (LPCTSTR objtype, LPCTSTR id, LPCTSTR param) = 0;
virtual   CMapStringToStringArray* GetObjectParamList(LPCTSTR objtype, LPCTSTR id, LPCTSTR param) = 0;
virtual   CStringArray*   GetObjectParamList(LPCTSTR objtype, LPCTSTR id, LPCTSTR param, LPCTSTR name) = 0;
virtual   void       GetObjectParams   (LPCTSTR objtype, LPCTSTR id, Msg& msg) = 0;
virtual   void       SetObjectParamInt (LPCTSTR objtype, LPCTSTR id, LPCTSTR param, int val) = 0;
virtual   CString GetObjectIdByParam(LPCTSTR type, LPCTSTR param, LPCTSTR val) = 0;
virtual   CString GetObjectIdByName(LPCTSTR type, LPCTSTR name) = 0;
virtual   CString GetObjectParentId(LPCTSTR objtype, LPCTSTR id, LPCTSTR parent) = 0;
virtual   int GetObjectIds(LPCTSTR objtype, CStringArray& list, LPCTSTR main_id = NULL) = 0;
```

```
        virtual   int GetObjectChildIds(LPCTSTR objtype, LPCTSTR objid, LPCTSTR childtype, CStringArray& list) = 0;
};
```

NissObjectDLLExt

```
class NissObjectDLLExt
{
protected:
        CoreInterface* m_pCore;
public:
        NissObjectDLLExt(CoreInterface* core) { m_pCore = core; }

        virtual CString GetObjectType() = 0;
        virtual CString GetParentType() = 0;
        virtual int GetPos()  { return -1; }
        virtual CString GetPort() { return CString(); }
        virtual CString GetProcessName() { return CString(); }
        virtual CString GetDeviceType() { return CString(); }

        virtual BOOL HasChild() { return FALSE; }
```

```
virtual UINT HasSetupPanel()  { return FALSE; }

virtual void OnPanelInit(CWnd*) {}

virtual void OnPanelLoad(CWnd*,Msg&) {}

virtual void OnPanelSave(CWnd*,Msg&) {}

virtual void OnPanelExit(CWnd*) {}

virtual void OnPanelButtonPressed(CWnd*,UINT) {}

virtual BOOL IsRegionObject()   { return FALSE; }

virtual BOOL IsProcessObject()  { return FALSE; }

virtual BOOL IsIncludeParentId()            { return 0; }

virtual BOOL IsWantAllEvents()                              { return 0; }

virtual CString DescribeSubscribeObjectsList() { return CString(); }

virtual CString GetIncludeIdParentType(){ return CString(); }

virtual CString DescribeParamLists(){ return CString(); }


virtual void OnCreate(Msg&) {}

virtual void OnChange(Msg&,Msg&) {}

virtual void OnDelete(Msg&) {}

virtual void OnInit(Msg&)           {}

virtual void OnEnable(Msg&)  {}

virtual void OnDisable(Msg&) {}

virtual BOOL OnEvent(Event&) { return FALSE; }
```

```
            virtual BOOL OnReact(React&) { return FALSE; }
};
```