Hardware and Software Module Integration Guide

1. Hardware and Software Module Integration Guide. Introduction 4
2. Integrating hardware and software modules with Intellect 4
2.1 General information on hardware and software modules integrating 4
2.2 Editing the DBI file 5
2.2.1 Adding Objects to Intellect.dbi 5
2.2.2 Using the ddi.exe Tool to Work with DBI files 8
2.3 Editing the DDI file 10
2.3.1 Adding Object Information to intellect.ddi 11
2.3.2 Using the ddi.exe Tool to Work with DDI files 13
2.4 Additional Functionality of the ddi.exe Utility 17
2.5 Creating MDL files 18
2.5.1 MDL File Creation Wizard 28
2.6 Creating RUN files 29
2.7 Creating and Configuring Integrated Objects (Modules) in Intellect 30
3. Intellect Integration Developer Kit (IIDK) 31
3.1 General Information on IIDK 32
3.1.1 Purpose of the IIDK 32
3.1.2 Developer Requirements 32
3.1.3 IIDK Components 32
3.2 Connecting to Intellect 33
3.2.1 Connection Parameters 33
3.2.2 IIDK Interface Object 33
3.2.3 Configuring passing events throung IIDK Interface object 34
3.2.4 Features of ATMs integration. ATM object 35
3.3 IIDK Functions 35
3.3.1 Connect 35
3.3.2 SendMsg 37
3.3.3 Disconnect 38
3.3.4 Other functions 38
3.4 Sent Message Syntax 41
3.4.1 Message Syntax 41
3.4.2 Message Syntax (port 900) 42
3.4.3 Using the Event and React classes 43
3.5 Examples of Managing System Objects 43
3.5.1 Adding, Updating, and Deleting System Objects 44
3.5.2 Working with the System in the Multiuser Mode 44
3.5.3 Determining Computers Where Intellect was Unloaded (via Port 1030) 45
3.5.4 Redirecting Video Cameras to the Monitor 45
3.5.5 Obtaining Object Parameters (via Port 1030) GET_CONFIG 45

3.5.6 Obtaining Information on Object States GET_STATE and GET_LIST 45
3.5.7 Showing Information Messages. SET_STATE 46
3.5.8 Live and archived video 46
3.5.9 Telemetry control 47
3.5.10 Map layer operations 47
4. Hardware and Software Module Integration Guide. Postscript 47
5. APPENDIX 1. DDI file structure 48
6. APPENDIX 2. NissObjectDLLExt and CoreInterface class declarations 49

Hardware and Software Module Integration Guide. Introduction

The Hardware and Software Module Integration Guide contains the information needed when integrating functional modules that perform the following tasks:

1. Adding new security hardware to the system.
2. 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 Hardware and Software Module Integration Guide of the online documentation.

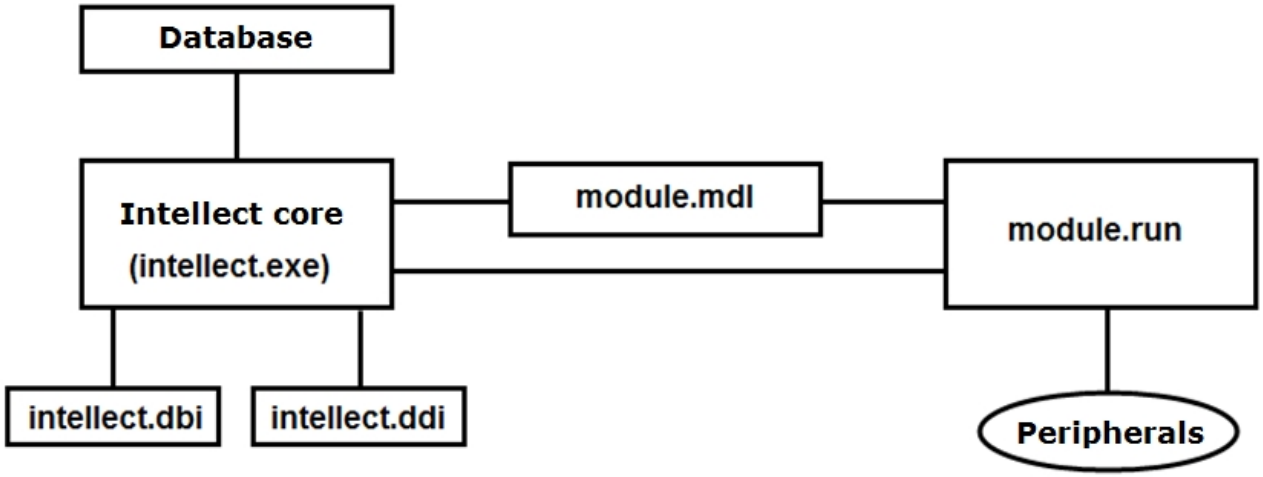
Integrating hardware and software modules with Intellect

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:

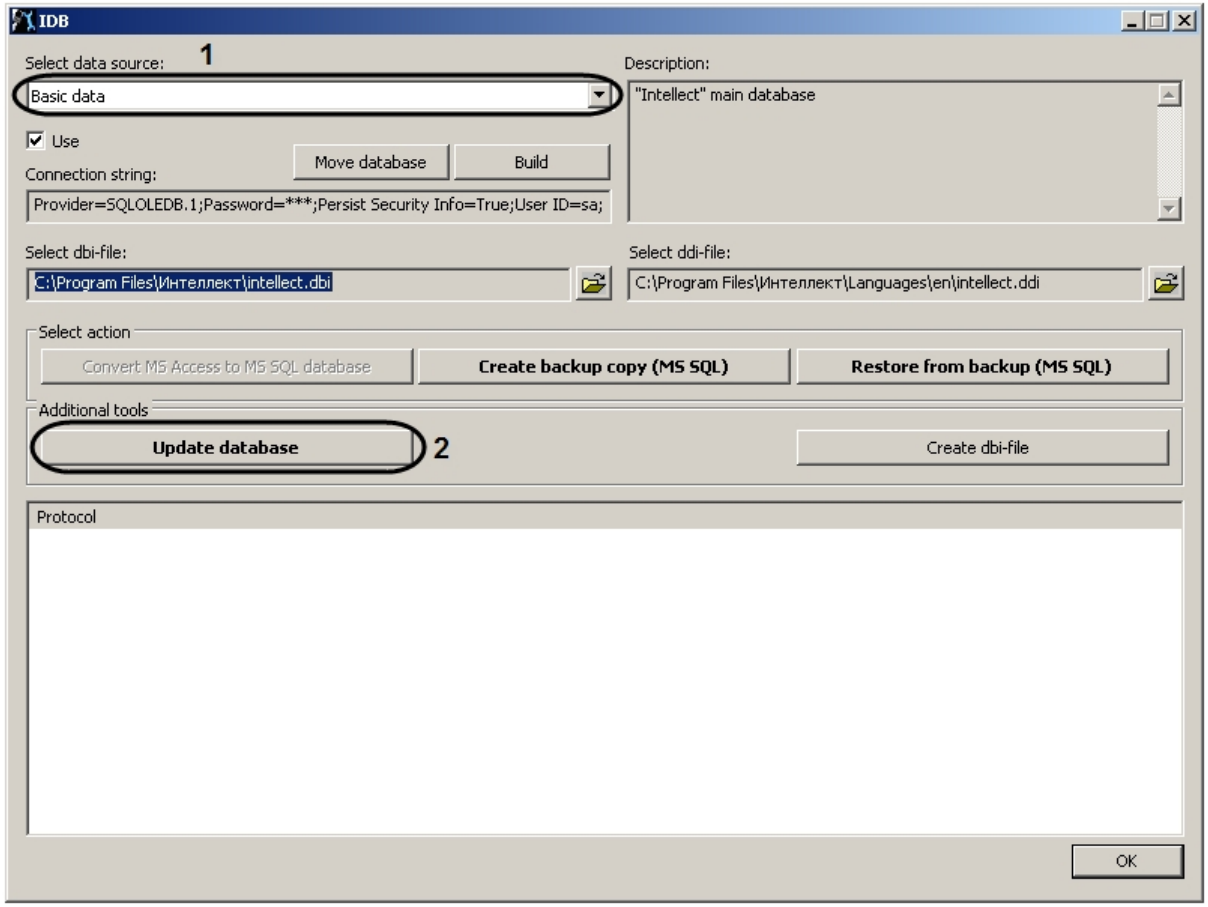
a. **port** – COM port address;
b. **address** – device address.

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

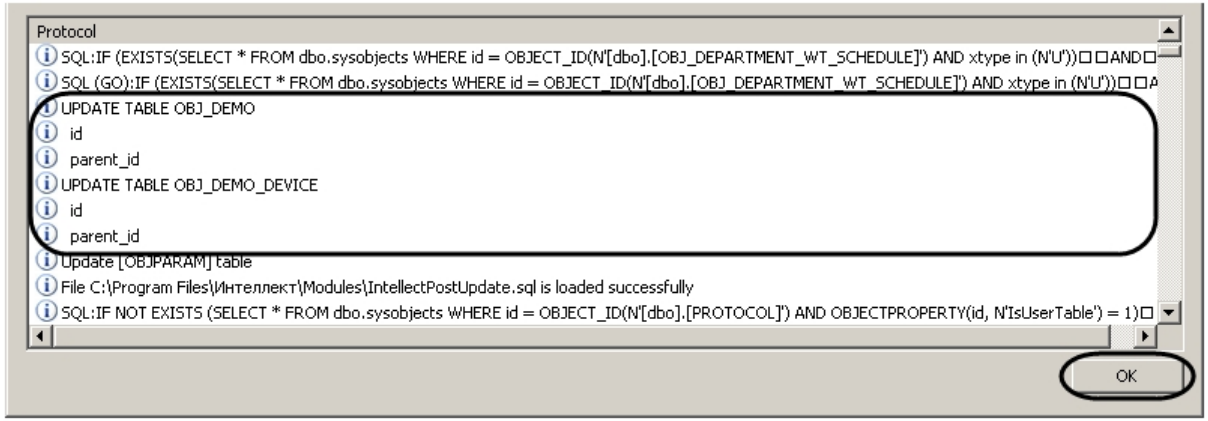
```
[OBJ_DEMO]
id, CHAR, 16
name, CHAR, 60
parent_id, CHAR, 16
flags, INTEGER
port, CHAR, 5

[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.

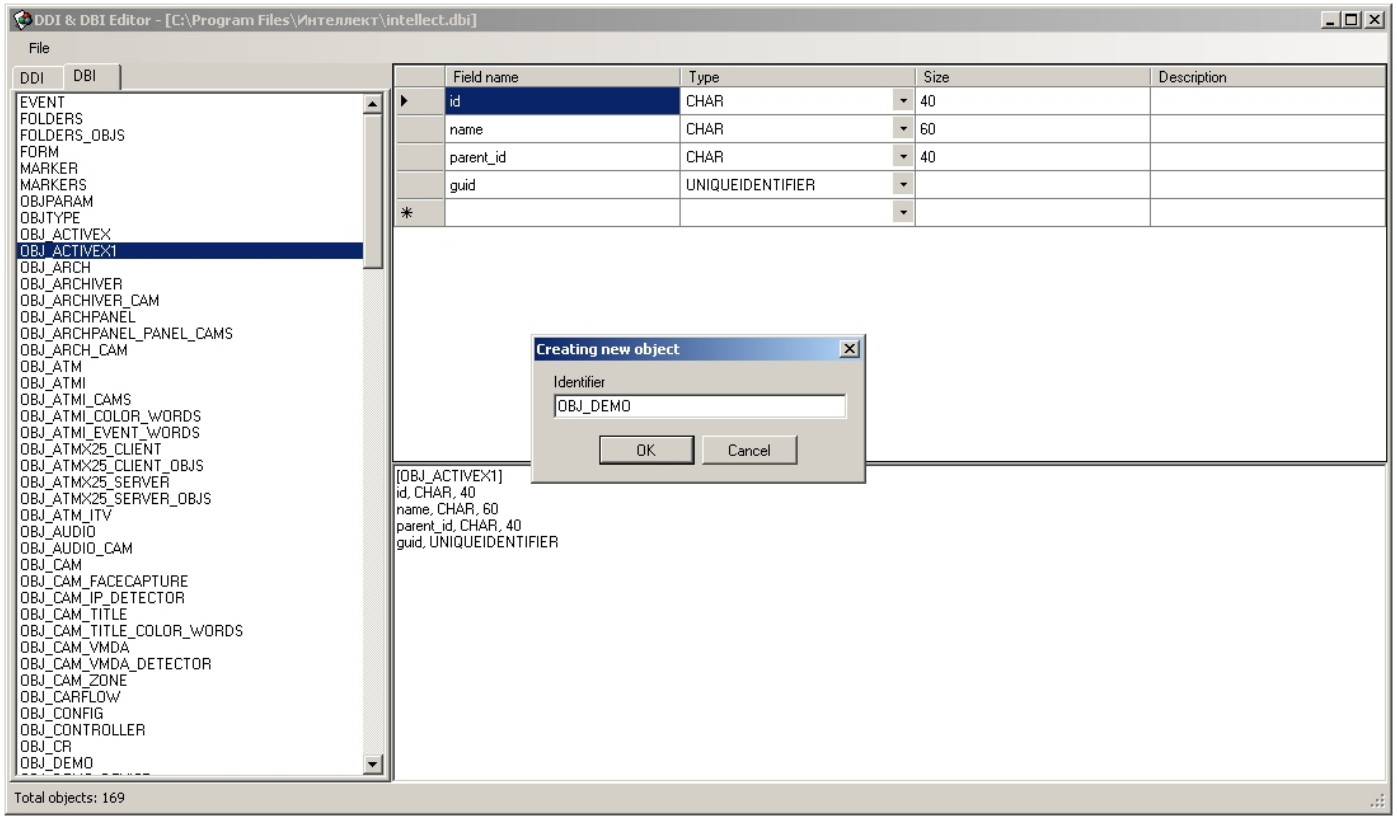
Using the ddi.exe Tool to Work with DBI files

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.**

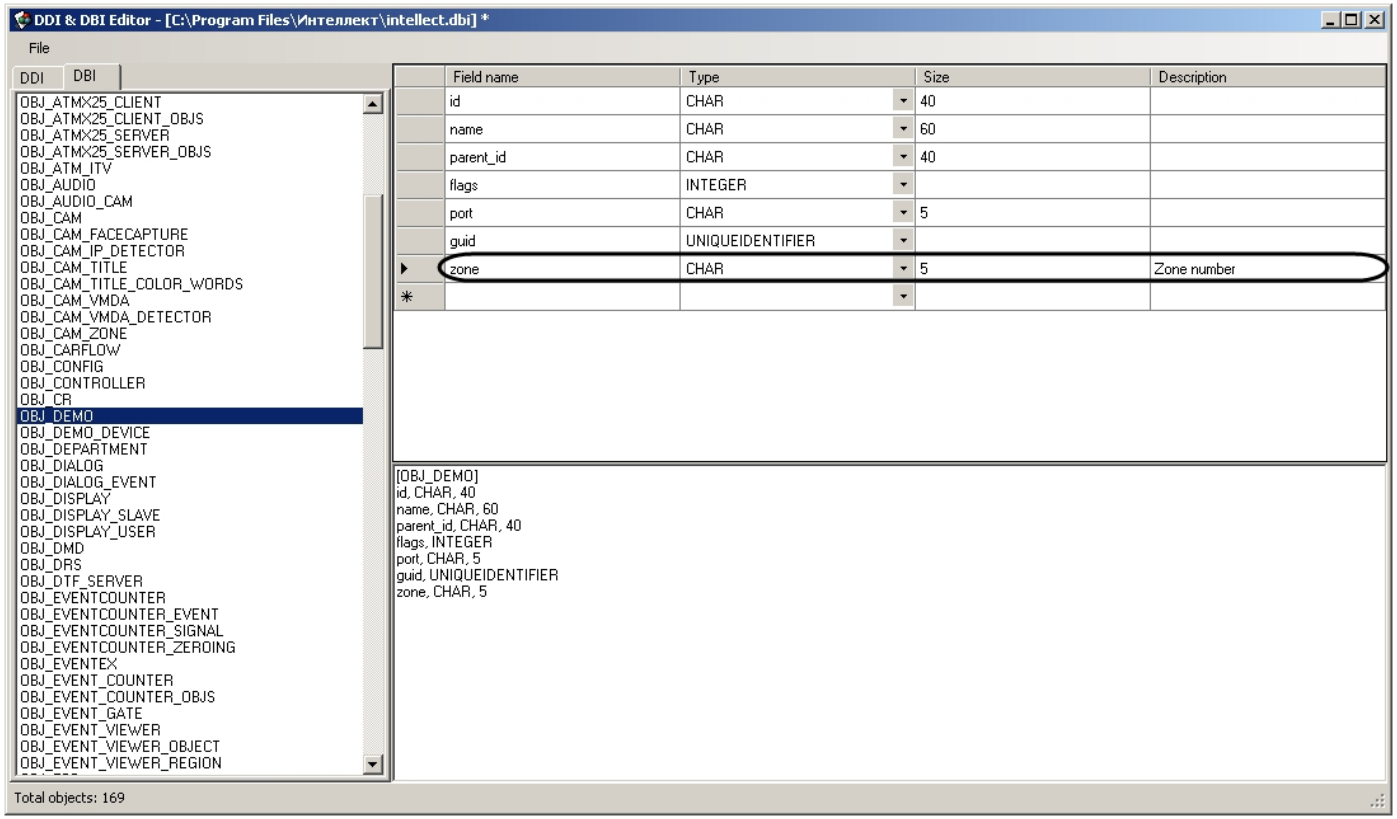


Note:
The mandatory fields are automatically added to the created object (see Section Adding Objects to Intellect.dbi).

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.

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.

```
<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.

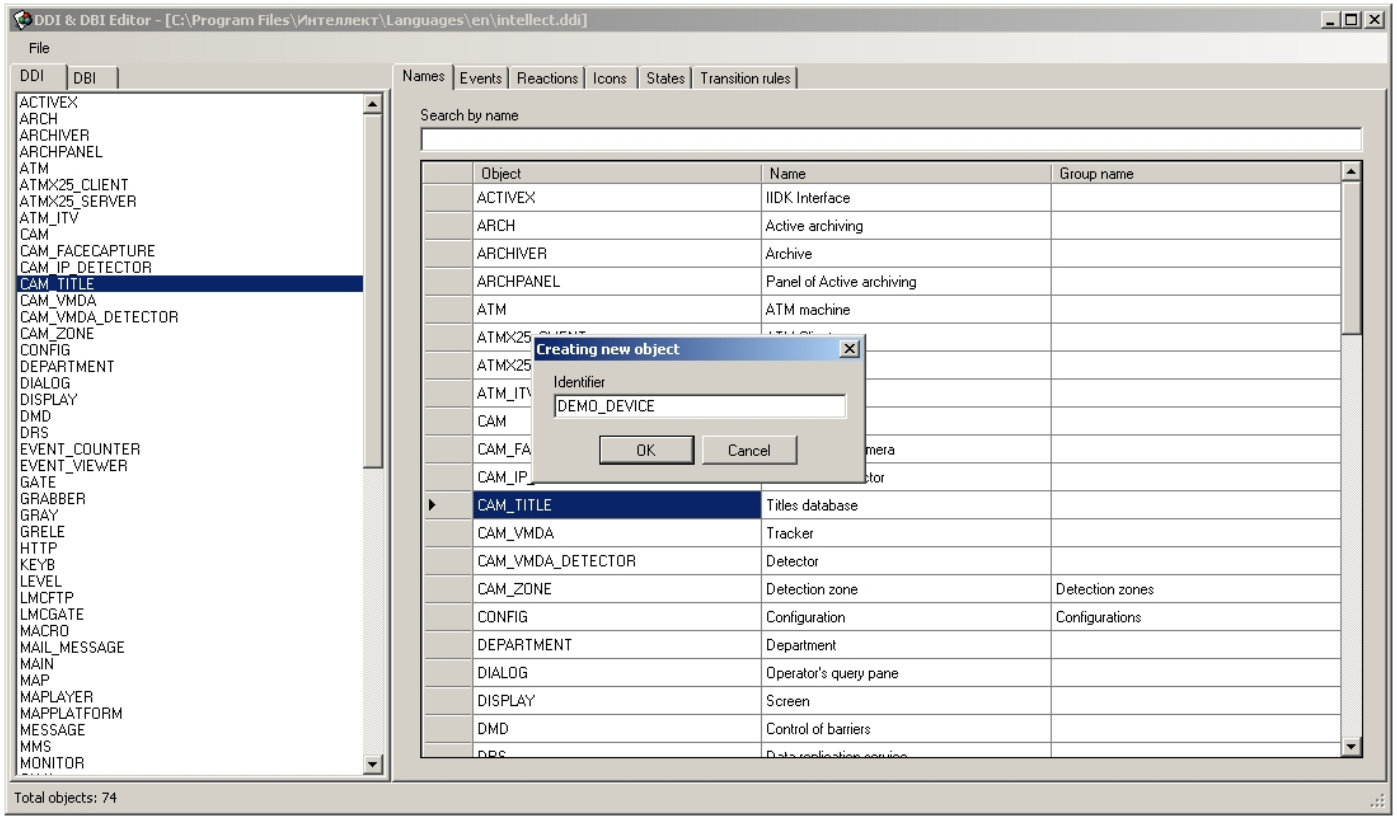
> **Attention!**
> After making changes to the DDI file, you must update the database structure by using idb.exe (see Steps 4–7 in Section Adding Objects to Intellect.dbi).

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.

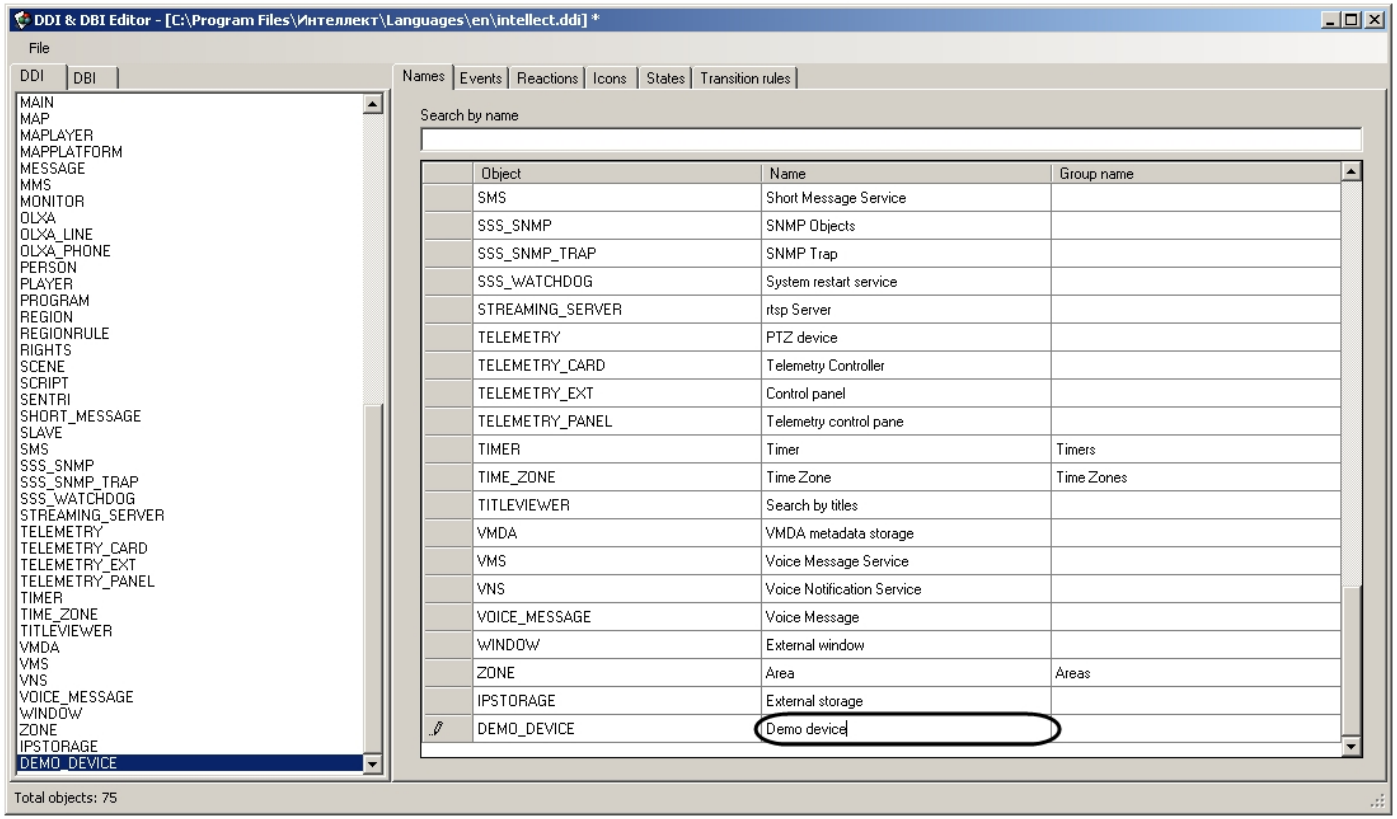
To add information on the **DEMO_DEVICE** object, do the following:

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**.

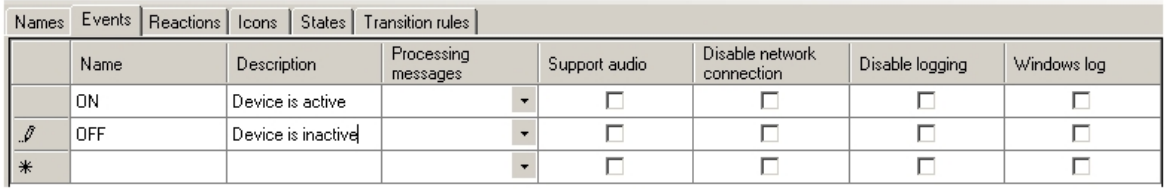


The DEMO_DEVICE object is now shown in the list of objects.

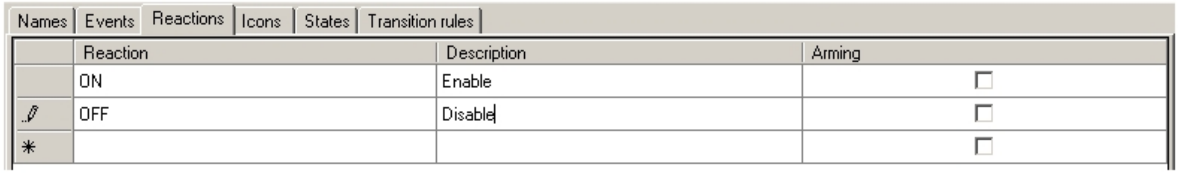
7. In the **Names** tab, enter an object name.



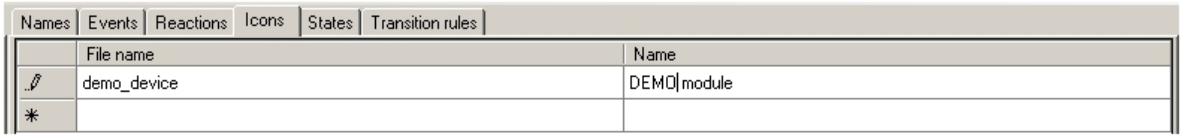
8. In the relevant tabs, add information on the DEMO_DEVICE object.
   a. In the **Events** tab, add the **ON** and **OFF** events.



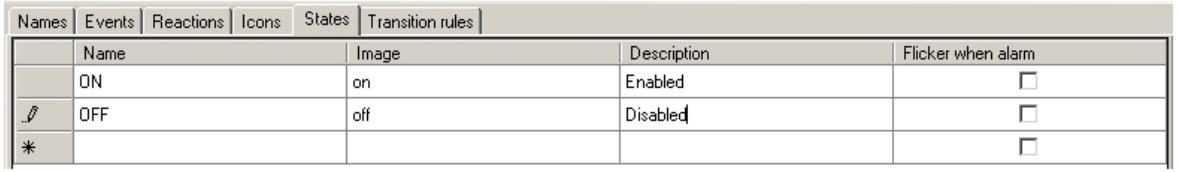
   b. In the **Reactions** tab, add the **ON** and **OFF** actions.



   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.**



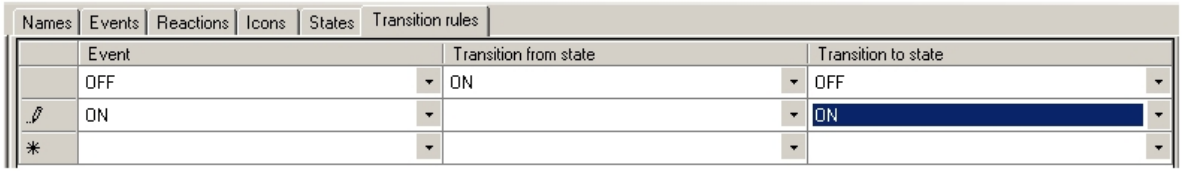
d. In the **States** tab, add the **ON** and **OFF** states. To show an object state on the **Map,** enter a BMP file name (the part that serves as an ID of the state).



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.



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.

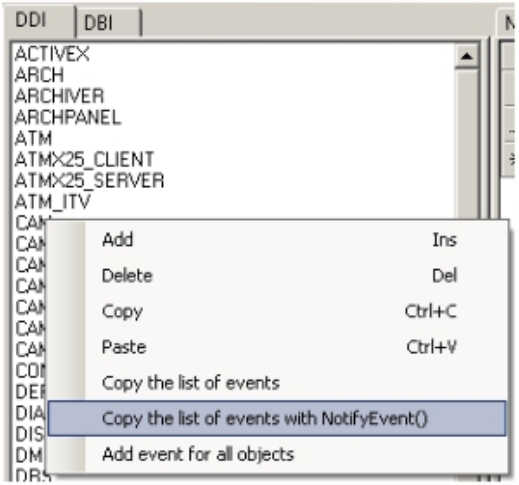
Note:
The fields of the ddi.exe tables are described in detail in APPENDIX 1. DDI file structure.

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).

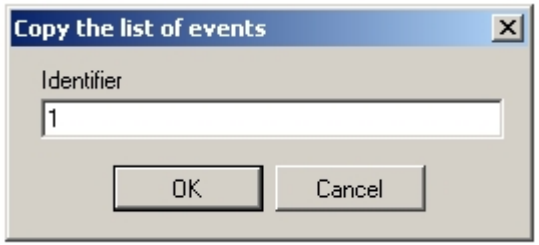
Additional Functionality of the ddi.exe Utility

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()**.



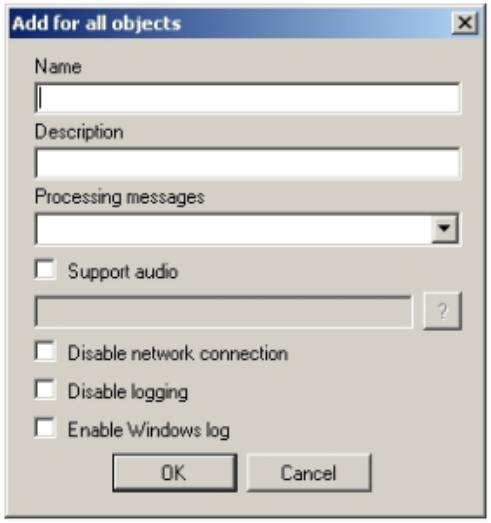
2. A dialog box opens. In the dialog box, enter the object ID to be used by the NotifyEvent function and click **OK.**



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.

Method	Description	Example
CoreInterface* m_pCore	A pointer to the core interface	
virtual BOOL IsWantAllEvents()	Returns TRUE if the OnEvent function receives events from all objects; returns FALSE if the function receives events from its own object only.	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
virtual CString DescribeSubscribeObjectsList()	The method accepts a comma-separated list of objects. When an object from the list is modified, the current object is notified.	
virtual CString GetObjectType()	Returns the object type	`virtual CString GetObjectType()` `{ return "DEMO"; }`
virtual CString GetParentType()	Returns the parent object type	`virtual CString GetParentType()` `{ return "SLAVE"; }`

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.	

<table>
<tr>
<td rowspan="2">virtual void OnPanelButtonPressed(CWnd*,UINT)</td>
<td>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</td>
<td>Virtual void OnPanelButtonPressed (CWnd* pwnd,UINT id)

{

if(id==IDC_TEST)

{ React react("DEMO",Id,"TEST");

m_pCore->DoReact(react); }

}</td>
</tr>
<tr>
<td>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.</td>
<td>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);</td>
</tr>
</table>

virtual BOOL IsRegionObject()	Shows whether the object supports Intellect's regions. Regions are used to group objects. They can also be used in the report system.	
virtual BOOL IsProcessObject()	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.*	

virtual void OnCreate(Msg&)	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.	virtual void OnCreate (Msg& msg)
		
		{
		
		msg.SetParam ("port","COM1");
		
		}

virtual void OnInit(Msg&)	Used when the object is initialized. The parameter is a reference to a message that contains object information.	virtual void OnInit (Msg& msg)
		
		{
		
		OnChange (msg, msg);
		
		}

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 Hardware and Software Module Integration Guide).

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:

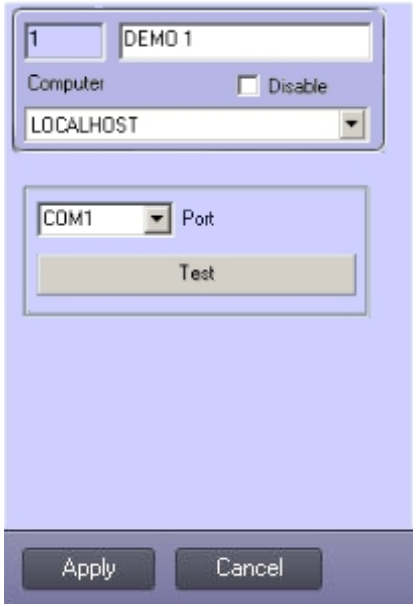
1. Copy the MDL and RUN files to the *Intellect\Modules* folder.
2. Start Intellect.
3. Under the **Computer** object, create the objects added earlier using the software module. For the *DEMO* module, Under the **Computer** object, create the **DEMO** object.

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

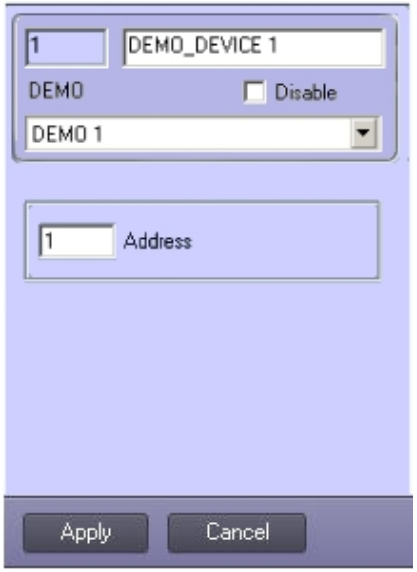
   

   As a result, the setup panels of the created objects become available.

   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.

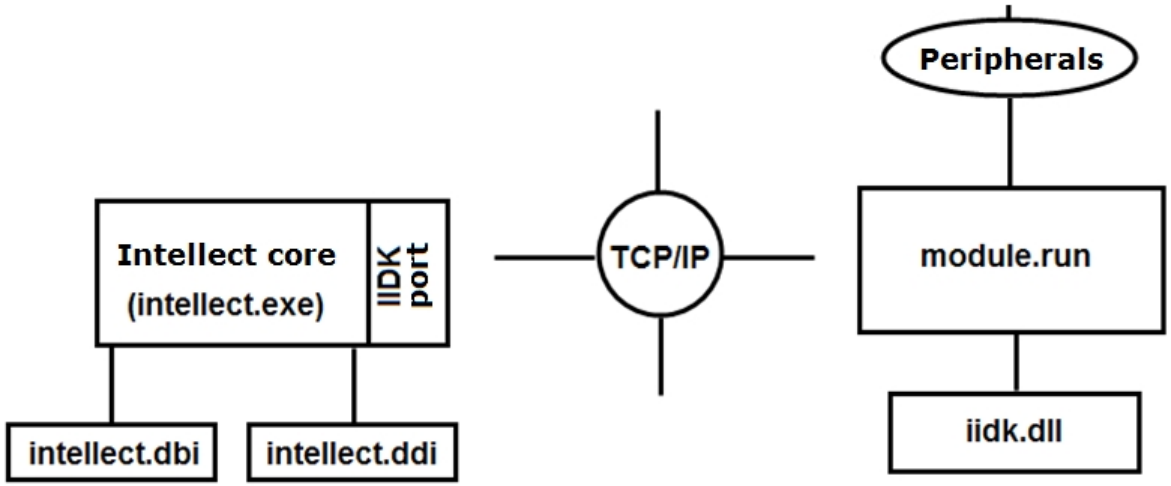
Intellect Integration Developer Kit (IIDK)

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 "").

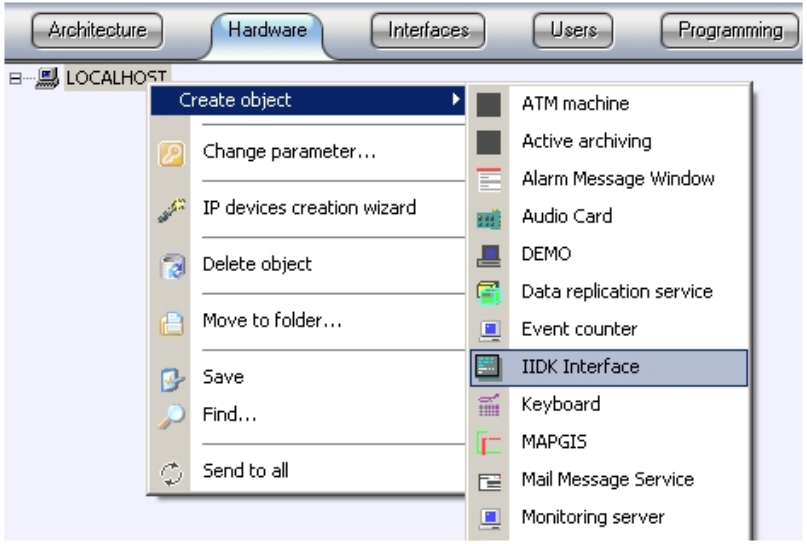
IIDK Interface Object

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

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

> **Note.**

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



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

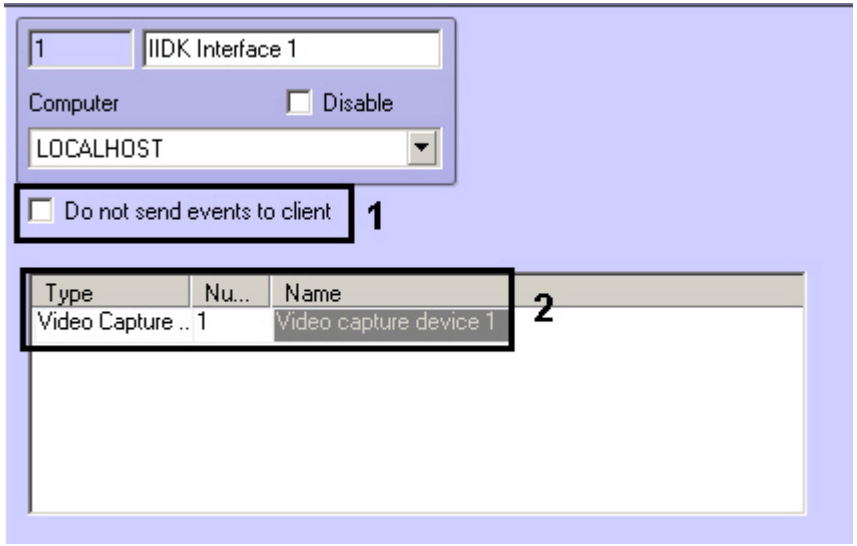
When Intellect's distributed architecture is used, the **IIDK Interface** object must be created on the computer that is running the software core (the core to which the connection is made). If the connection is made to a computer that has the *Operator Workstation* installed, the connection parameters 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.

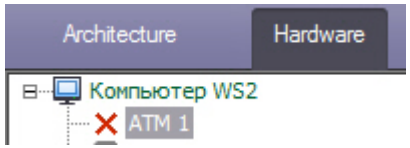


2. Set the **Do not send events to client** if it is not required to send any messages to client application (1).
3. In the table (2) specify list of objects, events from which have to be sent to connected client applications.

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** dial og 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	

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(LPCTSTR id);
```

Parameter	Description
id	Core connection ID

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 Hardware and Software Module Integration Guide.

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.

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 Hardware and Software Module Integration Guide.

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

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**

Hardware and Software Module 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.

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; }
};
```