Eocortex SDK

Published in 2026

Table of Contents

Published in 2026

Introduction

In previous versions, this document included a selection of Eocortex HTTP API requests. These requests have now been moved to the Eocortex REST API document, which is available for download on the Documentation for download page at doc.eocortex.com.

Published in 2026

General Information on Eocortex SDK

Eocortex SDK is a tool that allows you to develop software called plugins, which can extend functionality of existing **Eocortex** software system.

This tool is designed for .NET programmers who want to create plugins for **Eocortex**.

All source files of the tool and examples are coded for .NET in the C# language. **Microsoft Visual Studio** is assumed as a development environment. For understanding the document working knowledge of **Eocortex** terminology at the experienced user level is required. If necessary, you can refer to the operator and administrator instructions provided along with the **Eocortex**.

Since version 4.0, Eocortex uses Net6.
All plugins developed for versions 3.6 and earlier must be reworked to be compatible with the new platform version.

Within the **Eocortex SDK** framework, each plugin software is a descendant of one of the available base classes (interfaces) from the tools, and solves a specific range of tasks. Now the main base classes (interfaces) in the tools, which can be used by external software designers, are as follows:

Plugin Name	Name	Description
ExternalAction	Action	Base class that allows adding new actions for scripts and task scheduler
VideoAnalyst	Video analytics plugin	Base class used for video analytics on the server
PluginVisualiser	Visualizer plugin	Visualizer base class used for graphic display of specific information on the **Eocortex Client** application channel
IClientPluginMenuItem	Menu item plugin	Base class that allows to create proper sub-item in **Setup** menu of **Eocortex Client**
EventProcessor	Event processor plugin	Event processor base class that allows to register and generate its own events, get events from **Eocortex**, and execute commands in the channel. Plugins of this type are used to perform integration with other systems.
ICameraServiceProvider	Frame receiver plugin	IP devices' frame receiving interface that allows to get video, audio and motion detection data and control PTZ cameras.

All the above specified base classes (interfaces) types, as well as some other supporting entities, are the subjects of related chapters of the document. All the plugins exist and operate within the **Eocortex** channel. Thus, all plugin instances are isolated from each other by default, but, if necessary, relevant data can be shared within plugin static fields. As a rule, plugins that solve the same complex task are in one .NET assembly. Such assembly is a dynamic link library (DLL) which operates within **Eocortex**. Assemblies connection and plugin registration is carried out at the stage of startup of software system separate components (see Plugin registration in Eocortex).

Quick start – typical tasks

1. **IP camera integration**

   To connect an IP device (a camera), just implement the **Frame Receiver** plugin. All the necessary information about this type of plugin can be found in the Frame Receiver Plugin section. The plugin framework is in the folder with examples, in the **Camera.csproj** project.

2. **Integration with Access Control Systems, Security and Fire Alarm, POS terminals, etc.**

   The integration can be performed through the **Event Processor** plugin, which is able to receive events from **Eocortex**, generate its own events in **Eocortex** in the process of interaction with other systems, execute commands in **Eocortex** (recording on / off, setting presets, I/O from the cameras etc.) in the channel, getting access to **Eocortex** archive. For information about this plugin type, see Event processor Plugin section. The plugin example can be found in the examples folder in the **EventProcessor.cproj** project. If **Eocortex** is required only to receive video and audio, there is an easier option, discussed in the section HTTP interface for acquiring video; the example of acquiring video over HTTP is in the examples folder in the **HttpVideo.cproj** project.

3. **Video Analytics**

   Any video processing algorithm can be implemented using the **Video Analytics** plugin. All the analyst results are represented as events that can be further interpreted by **Menu Item** and **Visualizer** plugins. These plugins are discussed in sections Video Analytics Plugin, Visualizer Plugin and Menu Item Plugin; the example of usage can be found in the examples folder in the **Analyst.csproj** project.

Plugins

The present chapter deals with the process of plugin registration. It also discusses in detail each type of plugin. The most important code fragments are shown in the text.

Registration of plugins in Eocortex

As **Eocortex** software system separate components (**Server/Client/Configurator**, hereinafter - **host**) start up, .NET assemblies are searched in the **Plugins** folder of the starting application. The **IPlugin** interface must be implemented in each found assembly, as follows:

```
public interface IPlugin
{
        /// <summary>
        /// Returns unique module identifier
        /// </summary>
    Guid Id { get; }

        /// <summary>
        /// Returns module name
        /// </summary>
    string Name { get; }

        /// <summary>
        /// Returns module manufacturer name
        /// </summary>
        string Manufacturer { get; }

        /// <summary>
        /// Module initialization.
        /// It is called by the host during module registration in the system.
        /// </summary>
        /// <param name="host">Host interface</param>
      void Initialize(IPluginHost host);
 }
```

Example of the given interface implementation:

```
public class ModuleDef : IPlugin
{
    public Guid Id
    {
      get { return new Guid("17EE3457-8FC2-4C0F-B133-EF11D0C4F38C"); }
    }

    public string Name
    {
       get { return "Abandoned object detection"; }
    }

    public string Manufacturer
    {
        get { return "Eocortex"; }
    }

    public void Initialize(IPluginHost host)
    {
       host.RegisterAnalyst(typeof(AnalystExample));
       host.RegisterExternalEvent(typeof(ObjectLeftEvent));
```

```
    }
}
```

Once the specified interface has been detected by the host, the initialization member (**Initialize**) is called. As an argument **IPluginHost** interface, providing host service methods, is transferred to the initialization member:

```
public interface IPluginHost
{
       /// <summary>
       /// Gets log management interface
       /// </summary>
       /// <returns></returns>
       IMcLogMgr GetLogManager();

      /// <summary>
      /// Device registration.
      /// </summary>
   void RegisterDevType(DevType_RegInfo regInfo);

   /// <summary>
   /// Frame receiver registration.
   /// </summary>
   void RegisterRTFR(RTFR_RegInfo regInfo);

       /// <summary>
       /// Registers external event
       /// </summary>
       void RegisterExternalEvent(Type eventType);

      /// <summary>
      /// Registers external action
      /// </summary>
      void RegisterExternalAction(Type actionType);

      /// <summary>
      /// Registers a menu item in the Eocortex client
      /// </summary>
      void RegisterMenuItem(Type menuItemType, List<Guid> requiredPluginIDs);

// ... Other registration functions not shown here
}
```

Host service methods provide the following possibilities:

- Plugin actions logging with the **IMcLogMgr** interface, received by the **GetLogManager()** relevant method.
- Registration of plugins in the system for solving various problems.

Action plugin

An action plugin allows to extend the list of functions in scripts and in the task scheduler. It is necessary to make a class *ExternalAction* descendant to create this plugin type:

```
/// <summary>
/// Base action class.
/// </summary>
[Serializable]
public abstract class ExternalAction : IAction
{
   [NonSerialized]
```

```
    protected IActionHost actionHost;

    /// <summary>
    /// Initialization. It is called by host before starting work.
    /// </summary>
    /// <param name="host"></param>
    public virtual void Initialize(IActionHost host)
    {
        actionHost = host;
    }

    /// <summary>
    /// User action setting element. It is called by configurator.
    /// Must return UserControl type (WPF).
    /// </summary>
    public abstract object GetGUISettingsControl();

    /// <summary>
    /// Displays if the current action
    /// is configured properly
    /// </summary>
    public abstract bool IsConfigurated
    {
        get;
    }

    /// <summary>
    /// The feature shows if the action is related to
    /// the specific channel. In other words, whether the action affects
    /// the channel in any way.
    /// </summary>
    public virtual bool IsChannelIndependent
    {
        get
        {
          //the action is not related to channel in any way by default
          return true;
        }
    }

     /// <summary>
    /// Starting action to be executed
    /// </summary>
    public abstract void Run(RawChannelEvent channelEvent);

    /// <summary>
    /// Command execution in the channel. It is completed by server, can be used in
    /// the Run method (see above)
    /// </summary>
    [NonSerialized]
    public ExecuteCommandDelegate ExecuteCommand;
}
```

In the descendant class the following attributes must be created:

- **ActionGUIName** – name of action, used in graphic interface in the configurator.
- **GuidAttribute** – action identification.

ActionNeedsEventArgument attribute can be determined as an option to define that the event object must be transmitted from the server to call **Run** plugin method. It also means that the action is executed only at an event, and cannot be executed in the scheduled tasks when events do not occur in these tasks.

Also, the base class methods must be defined (redefined). Let's consider in detail the initialization member in which **IActionHost** interface is passed from the host. The interface is as follows:

```
/// <summary>
/// Interface providing the host capabilities
/// when initializing the action plugin.
/// </summary>
public interface IActionHost
{
    /// <summary>
    /// Getting information about the channel,
    /// in which the current action plugin
    /// runs.
    /// </summary>
    RawChannelInfo GetChannelInfo();

    /// <summary>
    /// Saves any serialized object in the
    /// Eocortex configuration. It is used in the configurator.
    /// </summary>
    /// <param name="id">Object identifier</param>
    /// <param name="obj">Object</param>
    void SaveObject(Guid id, object obj);

    /// <summary>
    /// Gets a previously serialized object from the configuration.
    /// It is used in the configurator.
    /// </summary>
    /// <param name="id"></param>
    /// <returns></returns>
    object GetObject(Guid id);
 }
```

The interface allows to get the information on the channel to which the plugin instance is attached. The information includes the channel name and identification. The identification must be used at command execution in the channel (ref. **ExecuteCommand** delegate).

For more detailed information on available commands refer to Section Event Processor Plugin. In addition, with the help of **SaveObject** and **GetObject** methods the given interface allows to save and download any serializable object using its identifier when **Eocortex** configuration is set. This technique allows to save and retrieve settings that are the same for all plugin instance configurations.

GetGUISettingsControl method must return the **UserControl** type element, which contains a graphic interface for the plugin instance configuration in the configurator. The whole graphic interface must be executed with WPF (Windows Presentation Foundation). **IsConfigurated** property shows if the action is configured correctly in the graphic interface. If anything is wrong, the configuration cannot be applied unless the mistakes are corrected.

For **Action** registration in the system, it is required to call **RegisterExternalAction** host interface method (see Plugin registration in Eocortex) when downloading the assembly and plugin in the **IPlugin** initialization class method.

Video Analytics Plugin

This plugin type analyses video stream frame-by-frame, and allows to create service detectors, such as abandoned objects detector, sabotage detector and others. To create this plugin type, the **VideoAnalyst** base class descendant must be created:

```
/// <summary>
/// Video analytics class. It is used for frame and motion map processing.
/// </summary>
public abstract class VideoAnalyst : IDisposable
{
   /// <summary>
   /// Analyst initialization. It is called by host before starting processing.
   /// </summary>
   /// <param name="Id">A channel identifier</param>
   /// <param name="archiveEventsReader">Archive access interface</param>
   /// <param name="mdZones">Motion detection zones.</param>
   /// <param name="settings">Analyst settings.</param>
   public abstract void Initialize(Guid Id, IArchiveEventsReader
   archiveEventsReader,
      List<MDZone> mdZones, PluginSettings settings);

   /// <summary>
   /// Frames and motion maps processing method. It is called by host.
   /// </summary>
   /// <param name="image">Frame</param>
   /// <param name="motionMap">A motion map, can be null</param>
   /// <param name="background">A motion detector background. Is equal to null, if
   /// NeedBackground == false</param>
   public abstract void Process(ImageData image, MotionMap motionMap,
      BackgroundImage background);

   /// <summary>
   /// Generates a previously registered external event in the channel.
   /// It is completed by host. It is called by analyst.
   /// </summary>
   public GenerateEventDelegate GenerateEvent;

   /// <summary>
   /// If the analyst supports work with partially decoded frames.
   /// True means that zoomed out video frames can be transmitted to the input,
   /// False means that video frames in original resolution are always transmitted
   /// to the input.
   /// </summary>
   public virtual bool SupportsPartlyDecodedFrames
   {
    get
    {
        return false;
    }
   }

   public virtual bool NeedBackground
   {
    get
    {
        return false;
    }
   }

   /// <summary>
   /// Pixel format supported
```

```
    /// </summary>
    public virtual VAPixelFormat PixelFormat
    {
     get
     {
         return VAPixelFormat.BGR24;
     }
    }

    /// <summary>
    /// Executes a command.
    /// </summary>
    /// <param name="cmdObj"></param>
    public abstract object ProcessCommand(object cmdObj);

    /// <summary>
    /// Replaces a motion detector for a preset one in the channel.
    /// Ii is completed by host. Can be null.
    /// </summary>
    public ReplaceMotionDetectorDelegate ReplaceMotionDetector;

    /// <summary>
    /// Resource deallocation
    /// </summary>
    public abstract void Dispose();

    /// <summary>
    /// Changes general or specific analyst settings, if necessary.
    /// A calling of method shall result in opening of user setting window.
    /// It is called by host (configurator).
    public abstract PluginSettings SetSettings(ISettingsHost settingsHost,
       PluginSettings settings);
}
```

Descendant class must redefine all base class abstract methods and, if necessary, virtual properties. Also, a subclass must contain **PluginGUINameAttribute**, which contains the name of the analyst displayed in **Eocortex Configurator** graphical shell. In case the analyst can be set up in the configurator, implementing **SetSettings** adjustment method and specifying **PluginHasSettingsAttribute** is required.

Video data is transmitted to the analyst input as a class described below:

```
public class ImageData
{
    public DateTime Timestamp;
    public byte[] Data;
    public System.Drawing.Size Size;
    public int Stride;
    public int BitPerPixel;
}
```

As each analytic plugin is attached by the user to a channel, **PluginSettings** object of configuration in **Eocortex** configurator consists of 2 parts:

1) Settings related to the current security channel
2) General analyst settings, unrelated to the selected security channel.

```
public struct PluginSettings
{
```

```
    /// <summary>
    /// Specific plugin settings related to the current channel. Can be null.
    /// A setting object must be serializable.
    /// </summary>
    public object channelSpecificSettings;

    /// <summary>
    /// General plugin settings. Are not related to the current channel. Can be null.
    /// A setting object must be serializable.
    /// </summary>
    public object generalSettings;
}
```

If analyst settings are unified for all channels, it is sufficient to complete only a general setting object. Otherwise, if all the analyst settings are related to a specific channel, it is sufficient to complete only a specific channel setting object. The objects of general and specific settings are user-defined. The only requirement for them is the possibility of their serialization, as all the analyst settings are kept in **Eocortex** general configuration.

Let us consider **ProcessCommand** method which allows the analyst to execute commands from the **Menu Item** plugin (see Menu Item plugin). This method receives a command as a serialized object formed on the **Menu Item** plugin side. As a result, this method shall also return the serialized object, which will be received subsequently and processed by the **Menu Item** plugin. This mechanism allows to implement client-server interaction, as the **Videoanalyst** plugin operates on the server, and the **Menu Item** plugin always operates in the client.

While processing video frames, the videoanalyst shall transmit the results of its analysis to the server by means of event generating. To do so, it is required to register in advance on the host side (see Plugin registration in Eocortex) the outside user event containing a description of all fields required for interpreting the operating results of the analyst. The event is registered by **IPluginHost** interface using **RegisterExternalEvent** method during the module initialization stage. The user event must inherit **RawChannelEvent** base class:

```
/// <summary>
/// Parent class in event hierarchy on channel.
/// </summary>
[Serializable]
public abstract class RawChannelEvent
{
    /// <summary>
    /// Event time stamp.
    /// </summary>
    public DateTime EventTime;

    /// <summary>
    /// An event comment.
    /// </summary>
    public string Comment;

    /// <summary>
    /// If event is local.
    /// Local events are not sent outside the current host.
    /// </summary>
    public bool IsLocal = false;

    /// <summary>
    /// If the event is to be saved in the database
    /// </summary>
    public bool Save = true;
```

```
    /// <summary>
    /// The event saving mode in the database.
    /// </summary>
    public abstract EventArchiveSaveMode SaveMode
    {
        get;
    }

    public RawChannelEvent()
    {
        EventTime = DateTime.UtcNow;
    }
}
```

Each user event should have a number of mandatory attributes:

- **GuidAttribute** – explicitly defines a unique event
- **Serializable** - allows to complete event serialization

In addition, there are following optional attributes:

- **EventNameGUI** - names the event in the host graphic interface. Unless the attribute is specified, the event is not displayed in scripts in the configurator;
- **EventNameDatabase** - a database table in which event instances are saved; the attribute must be used if the event has fields that must be saved in the database (it is important that **SaveMode** event property takes **Special** or **Both** values);
- **EventGeneratesAlarmByDefault** - a default event is an alarm, "**Generate alarm**" is automatically attached to the event when new channel is created in **Eocortex** configurator;
- **EventGenerationFrequency -** denotes event generation frequency, requires EventGenerationFrequencyMode (ref. below). The attribute is recommended as it affects server operation when events are saved in the database. Unless the attribute is not defined, Middle mode is used by default. This mode is preferred. Low mode is not advisable as events are recorded in the specific database, critical for functioning.

```
/// <summary>
/// Event generation frequency.
/// </summary>
public enum EventGenerationFrequencyMode
{
    /// <summary>
    /// Events are generated at frequency,
    /// close to the frequency of frame analysis
    /// </summary>
    High = 0,
    /// <summary>
    /// Events are generated at frequency compared to
    /// half of frame analysis frequency
    /// </summary>
    Middle
}
```

If the user event contains fields that must be saved in the database, the **EventFieldSaveable** attribute should be denoted for each field. The attribute needs a field ordinal number - **Order** (starting from 0) and **IsIndexable** flag, which denotes if the field index is required.

The following event field types with the following attributes **int**, **bool**, **long**, **double**, **DateTime**, **string**, **Guid**, **byte[]** are supported.

SaveMode property defines if the event is to be saved in the database. The event can be saved in a base table, that contains only **RawChannelEvent** class fields, and in a specific one with all event fields. It is possible to save all events in both tables. The base table allows to record the occurrence of the event in the system. Afterwards the user can read the events from the base table with **Eocortex** tools.

Thus, the user event can be implemented as follows:

```
[GuidAttribute("389EDCE2-54BB-4C2C-9984-51B7516A5DDF")]
[EventNameGUI("The object is left")]
[EventDatabaseName("objectleft")]
[EventGeneratesAlarmByDefault]
[EventGenerationFrequency(EventGenerationFrequencyMode.Low)]
[Serializable]
public class ObjectLeftEvent : RawChannelEvent
{
   [EventFieldSaveable(0, true)]
   [EventFieldNameGUI("Object")]
   private string objectName;

   public ObjectLeftEvent(string objectName)
   {
       this.objectName = objectName;
   }

   public override EventArchiveSaveMode SaveMode
   {
       get  { return EventArchiveSaveMode.Both; }
   }
}
```

For **VideoAnalyst** plugin registration, the host interface **RegisterAnalyst** method must be called at the stage of assembly and plugin loading in **IPlugin** interface using class initialization method (see Plugin registration in Eocortex).

Visualiser Plugin

This plugin allows to display information from events that are received by **Eocortex Client** channels in graphics (e.g., frames of moving objects, identified numbers, faces, etc.). For this plugin type, the **RTVisualiser** class descendant is to be created:

```
/// <summary>
/// Visualiser class.
/// </summary>
public abstract class RTVisualiser
{
   /// <summary>
   /// Panel for primitive, text and another channel information drawing.
   /// </summary>
   protected IDrawingPanel drawingPanel;

   /// <summary>
   /// Graphical elements container. Allows to deposit separate
   /// UserControl elements in the channel.
   /// </summary>
   protected Panel controlsContrainer;

   protected IPluginToolSet pluginToolset;

   /// <summary>
   /// Visualiser initialization. It is called by host.
   /// </summary>
```

```
    /// <param name="Id">A channel identifier</param>
    /// <param name="pluginToolset"> Archive access interface used for sending
    commands
    /// for event subscription in the system.</param>
    /// <param name="drawingPanel"> A drawing panel.</param>
    /// <param name="controlsContrainer"></param>
    public virtual void Initialize(Guid Id, IPluginToolSet pluginToolset,
          IDrawingPanel drawingPanel, Panel controlsContrainer)
    {
     this.pluginToolset = pluginToolset;
     this.drawingPanel = drawingPanel;
     this.controlsContrainer = controlsContrainer;
    }

    /// <summary>
    /// Visualiser event processing. Specific drawing of event results.
    /// </summary>
    /// <param name="channelId">A channel identifier</param>
    /// <param name="chEv">Event.</param>
    /// <param name="isAlarm">If event is alarm</param>
    public abstract void ProcessEvent(Guid channelId, RawChannelEvent chEv,
          bool isAlarm);

    /// <summary>
    /// Delegate for sub-item registration in channel pop-up menu
    /// in Eocortex client.
    /// It is completed by host. It is called by visualiser.
    /// </summary>
    /// <param name="item"></param>
    public RegisterChannelMenuItemHandler RegisterChannelMenuItem;

    /// <summary>
    /// Clearing all the visualizer drawings.
    /// </summary>
    public abstract void Clear();

    /// <summary>
    /// All resources deallocation. It is obligatory to call Release base class
    /// in reloaded descendant methods.
    /// </summary>
    public virtual void Release()
    {
     drawingPanel = null;
     controlsContrainer = null;
     pluginToolset = null;
     RegisterChannelMenuItem = null;
    }
}
```

Each **Visualiser** plugin must define **ProcessEvent** method, which receives events from the channel. All events are generated by **Eocortex** and other plugins. The plugin can visualize events of any type, and they can be filtered using **if** operator and **is** key word, for example:

```
if (chEv is CounterEvent)
{
    ...
}
```

Visualiser can register its subitem in the pop-up menu (right click on the channel in the **Eocortex Client**). This allows to alter visualiser logic according to the user preference. It is provided by **RegisterChannelMenuItem** delegate.

For Visualiser plugin registration, the host interface **RegisterRTVisualiser** method must be called at the stage of assembly and plugin loading in **IPlugin** interface using class initialization method (see Registration of plugins in Eocortex).

Menu Item plugin

The plugin of this type allows to create a proper graphic interface in **Eocortex Client**, and the user can call it by clicking the corresponding **Configuration** menu item.

Typical plugin application is organization of client-server interaction with other plugins. For example, the plugin can process results of analytical plugin operation on the server. **ClientMenuItem** class descendant must be created to make a menu item plugin:

```
/// <summary>
/// Menu item class. It is used in client for adding
/// sub-items on the Configuration button menu.
/// </summary>
public abstract class ClientMenuItem : IDisposable
{
    private bool isEnabled = true;

    /// <summary>
    /// Plugin initialization.
    /// </summary>
    /// <param name="toolSet"></param>
    public abstract void Initialize(IPluginToolSet toolSet);

    /// <summary>
    /// Menu item name.
    /// </summary>
    public abstract string Name
    {
        get;
    }

    /// <summary>
    /// If the menu item is enabled.
    /// </summary>
    public bool IsEnabled
    {
        get
        {
            return isEnabled;
        }
        set
        {
            isEnabled = value;
        }
    }

    /// <summary>
    /// Processing method of click (keystroke) of the user
    /// </summary>
    public abstract void OnClicked();

    /// <summary>
    /// Resourses release method.
```

```
    /// </summary>
    public abstract void Dispose();
}
```

The initialization method should be defined in the descendant class, which receives interface with service functions from the host (**Eocortex Client**). The functions allow to receive channel identifiers and their names in current configuration. In addition, they provide access to **Eocortex** archive and possibility to send commands to analytical plugins and receive results of their execution. It is possible to subscribe for all events occurring in the system. The interface is as follows:

```
/// <summary>
/// A host provided interface for
/// archive access, sending commands,
/// system events subscription.
/// </summary>
public interface IPluginToolSet
{
    /// <summary>
    /// Receives archive access interface
    /// </summary>
    /// <returns></returns>
    IArchiveEventsReader GetArchiveReader();

    /// <summary>
    /// Sends a command to plugin
    /// running in the server.
    /// </summary>
    /// <param name="pluginId">Plugin identifier</param>
    /// <param name="channelsId">Channel identifiers</param>
    /// <param name="cmdObj">Command</param>
    /// <returns> Returns each channel result.
      /// The key is channel identifier.
    /// Value is a result of the command execution.</returns>
    Dictionary<Guid, object> SendChannelsCommand(Guid pluginId,
            List<Guid> channelsId, object cmdObj);

    /// <summary>
    /// Installs/deletes the channel event handler.
    /// </summary>
    /// <param name="subscrId"> A subscriber identifier </param>
    /// <param name="channelId"> A channel identifier </param>
    /// <param name="eventsHandler"> Handler. If it is null, then
      /// previously installed handler is deleted.</param>
    void SetEventsHandler(Guid subscrId,
            Guid channelId, EventsHandler eventsHandler);

    /// <summary>
    /// Receives plugin settings with a specified identifier for the channel.
    /// </summary>
    /// <param name="channelId">A channel identifier</param>
    /// <param name="pluginId">Plugin identifier</param>
    /// <returns></returns>
    PluginSettings GetPluginSettings(Guid channelId, Guid pluginId);
}
```

GetArchiveReader method provides access interface to archive and current channels information in the configuration. Detailed data about this interface are kept in **Eocortex SDK** source codes.

SendChannelsCommand method sends commands to different plugin instances of the same type according to the list of channels and receives results of command execution. **SetEventsHandler** method sets up and deletes the channel event handler.

OnClicked method must implement the user logic through graphic interface. It is called by client by clicking on the plugin-related menu item.

For **Menu Item** plugin registration, the host interface **RegisterMenuItem** should be called at the stage of assembly and plugin loading in **IPlugin** interface of class initialization method (see Plugin registration in Eocortex).

Event Processor plugin

The **Event Processor** plugin allows to receive and process signals from external systems. While processing signals, the plugin can execute commands and generate events in its channel. The event processor is created by means of inheritance from **EventProcessor** base class:

```
/// <summary>
/// Plugin class for system event processing, command and proper event generating.
/// </summary>
public abstract class EventProcessor
{
   /// <summary>
   /// Initialization. Called by host.
   /// </summary>
   /// <param name="archiveReader">Archive access interface</param>
   /// <param name="channelSpecificSettings">Plugin settings</param>
   public abstract void Initialize(IArchiveEventsReader archiveReader,
         PluginSettings settings);

   /// <summary>
   /// Generates a registered external event in the channel. Is completed
   /// by host. Called by plugin.
   /// </summary>
   public GenerateEventDelegate GenerateEvent;

   /// <summary>
   /// Sends a set command to be executed in the channel. Is completed by host.
    /// Called by plugin.
   /// </summary>
   public ExecuteCommandExDelegate ExecuteCommand;

   /// <summary>
   /// Allows to subscribe for events on the channel. Completed by
    /// plugin if necessary at the initialization stage.
   /// </summary>
   public ReceiveEventDelegate OnChannelEventReceived;

   /// <summary>
   /// Changes general or specific settings, if necessary.
   /// Method calling must result in user setting window opening.
   /// It is called by host (configurator).
   /// </summary>
   /// <param name="settings"> Current plugin settings.</param>
   /// <returns>New plugin settings.</returns>
   public abstract PluginSettings SetSettings(PluginSettings settings);
}
```

A subclass must contain an obligatory attribute **PluginGUINameAttribute**, which has the analyst name displayed in **Eocortex Configurator** graphics. In case the analyst can be set up in the configurator, **SetSettings** adjustment method must be implemented and **PluginHasSettingsAttribute** identified.

Initialize method receives the interface for access to **Eocortex** archive from the host and plugin setting object. As each plugin is attached to a channel, **PluginSettings** object of configuration consists of 2 parts:

1) Settings related to the current security channel;
2) General analyst settings, unrelated to the selected security channel.

```
public struct PluginSettings
{
    /// <summary>
    /// Specific plugin settings related to the current channel. Can be null.
    /// A setting object must be serializable.
    /// </summary>
    public object channelSpecificSettings;

    /// <summary>
    /// General plugin settings. Not related to the current channel. Can be null.
    /// A setting object must be serializable.
    /// </summary>
    public object generalSettings;
}
```

If the analyst settings are unified for all channels, it will be enough to complete only general settings object. Otherwise, if all the plugin settings are related to a single channel, it will be enough to complete only a specific channel settings object. General and specific objects are user-defined. The only requirement is the object serialization, as all the plugin settings are kept in **Eocortex** general configuration.

If the plugin must execute a specific command as a result of its operation (such as, turn recording on/off, set up a preset on a camera, turn a camera, etc.), a command object should be created. Currently there are following commands:

- **RawEnableRecordingCommand** - turns on a record in the channel, record interval is defined optionally;
- **RawDisableRecordingCommand** - turns off a record in the channel;
- **RawGoToPresetPtzCommand** - sets up a preset on a camera;
- **RawGoHomePtzCommand** - sets up home camera position;
- **RawStopPtzCommand** - stops ptz (panorama/tilt/zoom) command execution on a camera;
- **RawMovePtzCommand** - one step movement;
- **RawZoomPtzCommand** – relative approximation (zoom);
- **RawStartMovePtzCommand** - continuous (preferred flowing) motion;
- **RawMoveToPtzCommand** - turns a camera so that a reference point is in the center of the frame area;
- **RawSetOutputIOCommand** - adjusts a signal level on the camera output;
- **RawSetOutputPulsesIOCommand** - generates the impulse sequence on the camera output

The **Event Processor** plugin can subscribe for events occurring at the channel. To do so the event processor has to complete **OnChannelEventReceived** delegate when initializing. In addition, the plugin can generate its own events at the channel.

For the **Event Processor** plugin registration, the host interface **RegisterEventProcessor** should be called at the stage of assembly and plugin loading in **IPlugin** interface of class initialization method (see Plugin registration in Eocortex).

Frame receiver plugin

The plugin of this type allows to receive video and sound from IP devices and motion detection data, to control PTZ cameras and IP devices' inputs/outputs (IO). To perform the subtask of receiving frames, the **IRealTimeFrameReceiver** interface must be implemented:

```
/// <summary>
/// Interface of receiving real time frames.
/// Used for creation of plugins that receive frames
/// from IP cameras.
/// </summary>
public interface IRealTimeFrameReceiver
{
    /// <summary>
    /// Event handler on receiving a new frame.
    /// It is called by the side that operates interface.
    /// The host is subscribed for events.
    /// </summary>
    event NewRawFrameEventHandler NewRawFrame;

    /// <summary>
    /// Event handler. It is called by the side that operates interface.
    /// The host is subscribed for the handler.
    /// </summary>
    event NewRawEventHandler NewEvent;

    /// <summary>
    /// Handler of a new record entering in the log. Informs the host that
    /// Connection Log field has changed (ref. below).
    /// The log is viewed in the Configurator after clicking the "Log History"
    /// It is called by the side that operates interface.
    /// The host is subscribed for the handler.
    /// </summary>
      event EventHandler NewLogRecord;

    /// <summary>
    /// A flag that denotes if it is necessary to enter a record in log.
    /// It is changed by host.
    /// </summary>
      bool IsWritingConnectionLog
      {
        get;
        set;
      }

    /// <summary>
    /// Current content of the connection log.
    /// </summary>
      string ConnectionLog
      {
        get;
      }

    /// <summary>
    /// If the stream is active
    /// </summary>
    /// <param name="streamType">Stream type</param>
    /// <returns></returns>
    bool IsStreamActive(ChannelStreamTypes streamType);

    /// <summary>
    /// Starts the specified stream to receive frames
```

```
    /// </summary>
    /// <param name="streamType"></param>
    void StartStream(ChannelStreamTypes streamType);

    /// <summary>
    /// Stops the specified stream
    /// </summary>
    /// <param name="streamType"></param>
    void StopStream(ChannelStreamTypes streamType);

    /// <summary>
    /// Sends sound to the device (duplex sound realization).
    /// </summary>
    /// <param name="soundData"></param>
    void SendSound(byte[] soundData);

    /// <summary>
    /// Releases all resources. Closes all streams.
    /// </summary>
    void Release();
}
```

While implementing this interface, it is necessary to create a mechanism for working with data streams, which can be a sequence of particular frame type, or a sequence of events. Current **Eocortex SDK** version contains the following data stream types:

```
/// <summary>
/// Channel stream types.
/// </summary>
public enum ChannelStreamTypes : ulong
{
    /// <summary>
    /// Main video stream
    /// </summary>
    MainVideo = 1,

       /// <summary>
       /// Alternative video stream
       /// </summary>
       AlternativeVideo = 2,

    /// <summary>
    /// Camera sound stream.
    /// </summary>
    MainSound = 4,

       /// <summary>
       /// Alternative sound stream
       /// </summary>
       AlternativeSound = 8,

    /// <summary>
    /// Reverse sound stream.
    /// </summary>
    OutputSound = 16,

    /// <summary>
    /// Motion detection data stream.
    /// </summary>
    MotionDetection = 32,

              /// <summary>
```

```
            /// Camera I/O system data stream
            /// </summary>
            IO = 64,
}
```

The data streams of different types are to be generated depending on the device capabilities and the channel settings in the configurator.

MainVideo and **AlterntaiveVideo** data streams consist of **RawVideoFrame** video frames, received from the camera.

MainSound and **AlternativeSound**, and **OutputSound** data streams consist of **RawSoundFrame** sound frame sequence. **MotionDetection** data stream consists of **RawChEv_MDresults** and **Raw ChEv_NoDetection** events sequence.

I/O stream consists of **RawChEv_InputSignalLevelChanged** events.

StartStream method starts the data streams from the host, in which the next stream is initialized. The method has to immediately return control to the host and implement long-term operations (input/output operations) in separate streams. Stopping the streams and control of their activities using **StopStream** and **IsStreamActive** methods is called by the host, and these actions must be completed immediately without any long-term operations. Streams return the results of their operation to the host by calling frame (**NewRawFrame**) and event (**NewEvent**) handlers.

For example, if the IP-device sends MJPEG frames, **MainVideo** (or **AlternativeVideo**), the data stream must call **NewRawFrame** and transmit **RawMJPEGFrame** video frame as one of the arguments:

```
/// <summary>
/// MJPEG frame
/// </summary>
 [Serializable]
public class RawMJPEGFrame : RawVideoFrame
{
     public RawMJPEGFrame() { }

     /// <summary>
     /// The frame data
     /// </summary>
     /// <param name="data"></param>
     public RawMJPEGFrame(byte[] data)
     {
           Data = data;
     }
}
```

Likewise, for **MainSound** (or **AlternativeSound**) data stream and sound in the G.711U format, the frame **RawG711UFrame** is to be transmitted:

```
/// <summary>
/// G.711U frame
/// </summary>
[Serializable]
public class RawG711UFrame : RawSoundFrame
{

     /// <summary>
     /// The frame data
     /// </summary>
     /// <param name="data"></param>
        public RawG711UFrame(byte[] data)
        {
             this.Data = data;
             this.samplesRate = 8000;
```

```
            this.bitsPerSample = 16;
            this.channels = 1;
            this.bitrate = 64000;
        }
}
```

After the creation of the appropriate type video frame, completing the following fields of **RawFrame** base class is required:

- **Id** – the frame identifier consisting of 2 parts: a sequence identifier (is generated at random before receiving the data stream) and a frame ordinal number.
- **Timestamp** – the frame timestamp, defined in UTC format.

In MPEG-4 and H.264 codecs:

- It is obligatory for P-frames to complete **Dependencies** field containing all frame identifiers on which the given frame depends.
- The decoder initializing information is to be defined in **SpecInitData** field in each I-frame.

Example of MJPEG frame creating:

```
RawMJPEGFrame jpegFrame = new RawMJPEGFrame(frameData);
jpegFrame.Id.SeqId = videoSeqId;
jpegFrame.Id.NumInSeq = videoNumInSeq++;
jpegFrame.TimeStamp = DateTime.UtcNow;
```

where the initial settings are:

```
// video frame sequence identifier
Guid videoSeqId = Guid.NewGuid();
// Next video frame number in the sequence
long videoNumInSeq = 0;
```

Example of H.264 I frame:

```
RawH264_I_Frame iFrame = new RawH264_I_Frame(frameData);
iFrame.Id.SeqId = videoSeqId;
iFrame.Id.NumInSeq = videoNumInSeq++;
iFrame.TimeStamp = DateTime.UtcNow;
iFrame.SpecInitData = initData;
```

where **initData** is the initializing information for a decoder.

Example of H.264 P frame:

```
RawH264_P_Frame pFrame = new RawH264_P_Frame(frameData);
pFrame.Id.SeqId = videoSeqId;
pFrame.Id.NumInSeq = videoNumInSeq++;
pFrame.TimeStamp = DateTime.UtcNow;
pFrame.Dependencies = frameDependencies;
```

where **frameDependencies** is the array of frame identifiers the frame depends on. In other words, it is a set of frames that must be decoded before the frame decoding.

Example of G.711U frame:

```
RawG711UFrame g711Frame = new RawG711UFrame(frameData);
g711Frame.Id.SeqId = audioSeqId;
g711Frame.Id.NumInSeq = audioNumInSeq++;
g711Frame.TimeStamp = DateTime.UtcNow;
```

If during the receiving of data streams a connection with the IP device becomes lost, **Frame Receiver** plugin must inform the host by generating **RawChEv_NoDataConnection** event with the obligatorily completed **StreamTypes** field which shows streams with the lost connection).

For registering frame receiver in the system, it is necessary to complete **DevType_RegInfo** registration information of the device and the **RTFR_RegInfo** plugin registration information. **DevType_RegInfo** class is described below:

```
/// <summary>
/// The device registration information.
/// is used during frame receiving plugin
/// registration.
/// </summary>
public class DevType_RegInfo
{
    /// <summary>
    /// Device identifier.
    /// </summary>
    public Guid DeviceTypeGuid;

    /// <summary>
    /// Name of manufacturer.
    /// </summary>
    public string DevTypeBrandName;

    /// <summary>
    /// Device name.
    /// </summary>
    public string DevTypeModelName;

    /// <summary>
    /// List of device capabilities.
    /// </summary>
    public DevType_Capabilities Capabilities;

    /// <summary>
    /// List of available resolutions for the device.
    /// </summary>
    public List<VideoResolutions> AvailableResolutions = new
                                                    List<VideoResolutions>();

    /// <summary>
    /// Delegate that allows host to change camera/ video server settings.
      /// </summary>
      public SetDeviceParametersDelegate SetDeviceParameters;

      /// <summary>
      /// Receiving IRealTimeFrameReceiver interface.
      /// </summary>
      public GetRTFRDelegate GetRTFR;

    /// <summary>
    /// Receiving PTZ interface.
    /// </summary>
    public GetPtzControllerDelegate GetPtzController;

      /// <summary>
      /// Receiving I/O interface.
      /// </summary>
      public GetIOControllerDelegate GetIOController;
}
```

GetRTFR delegate, called by host, must return a specific implementation of **IRealTimeFrameReceiver** interface. The delegate receives connection parameters (**ConnectionParameters**) and substream parameters (**SubStreamParameters**), which have been defined by the user in the channel configuration as arguments.

Capabilities of IP device with which the **Frame Receiver** plugin operates are described in the **Capabilities** field:

```
/// <summary>
/// List of device capabilities
/// </summary>
public enum DevType_Capabilities : ulong
{

      /// <summary>
    /// Device supports only cameras.
    /// </summary>
    SupportsCameras = 1,
    /// <summary>
    /// Device supports cameras and video servers.
    /// </summary>
    SupportsCamerasAndServers = 2,
    /// <summary>
    /// Device supports alternative stream.
    /// </summary>
    SupportsAlternativeVideoStream = 4,
    /// <summary>
    /// Parameters (resolution, fps, compression), described by
       /// SupportedDeviceParameters/SupportedExtraParameters arrays, are
      independent of the format
       /// of the stream (are the same for mjpeg, mpeg4, h264).
    /// </summary>
    DeviceParametersFormatIndependent = 8,
}
```

If it is required to solve a task of controlling a tilting camera or its inputs/outputs (IO), the **IPtzController** and/or **IIOController** interfaces need to be implemented:

```
/// <summary>
/// Unified interface of tilting camera control implementation.
/// </summary>
public interface IPtzController
{
    #region ----------------- BASE PART -----------------

    /// <summary>
    /// A camera initialization.
    /// </summary>
    /// <returns></returns>
    void Initialization();

    /// <summary>
    /// Returns the camera capabilities.
    /// </summary>
    /// <returns></returns>
    PtzCapabilities GetCapabilities();

    /// <summary>
    /// Returns names of camera presets.
    /// The number of elements in resulting array corresponds to the number of
    /// presets.
    /// Each preset corresponds to a number equal to the array index.
    /// </summary>
```

```
    /// <returns>Names of camera presets.</returns>
    string[] GetPresetsNames();

    /// <summary>
    /// Defines a preset by its index.
    /// </summary>
    /// <param name="presetIndex"> The preset index.</param>
    void SetPresetPosition(int presetIndex);

    /// <summary>
    /// Moves the camera to the home position.
    /// </summary>
    void MoveToHome();

    /// <summary>
    /// Stops any PTZ command executing.
    /// </summary>
    void Stop();

    /// <summary>
    /// One step movement.
    /// </summary>
    /// <param name="panSpeed"> Horisontal speed. Range from -100 to
    /// 100.</param>
    /// <param name="tiltSpeed">Vertical speed. Range from -100 to
    /// 100.</param>
    void StepMove(int panSpeed, int tiltSpeed);

    /// <summary>
    /// Continuous (preferred flowing) motion.
    /// </summary>
    /// <param name="panSpeed">Horisontal speed. Range from -100 to
    /// 100.</param>
    /// <param name="tiltSpeed">Vertical speed. Range from -100 to
    /// 100.</param>
    void ContiniousMove(int panSpeed, int tiltSpeed);

    /// <summary>
    /// Relative zooming in.
    /// </summary>
    /// <param name="step">Step from 1 to 100.</param>
    void StepZoomIn(int step);

    /// <summary>
    /// Relative zooming out.
    /// </summary>
    /// <param name="step">Step from 1 to 100.</param>
    void StepZoomOut(int step);

    /// <summary>
    /// Continuous (preferred flowing) zooming in.
    /// </summary>
    /// <param name="speed">Speed. Range from 1 to 100.</param>
    void ContiniousZoomIn(int speed);

    /// Continuous (preferred flowing) zooming out.
    /// </summary>
    /// <param name="speed">Speed. Range from 1 to 100.</param>
    void ContiniousZoomOut(int speed);

    /// <summary>
    /// Returns the camera max. zooming in.
```

```
    /// </summary>
    /// <returns>The camera max. zooming in. If the function is not
    /// supported, a negative number is returned.</returns>
    double GetMaxZoomFactor();

/// <summary>
    /// Returns the current camera zooming in.
    /// </summary>
    /// <returns> Current camera zooming in. If the function is not supported,
    /// a negative number is returned.</returns>
    double GetCurrentZoomFactor();

    /// <summary>
    /// Sets up absolute zooming in. No operation if camera does not support
    /// the function. Ref. PtzCapabilities.
    /// </summary>
    void SetZoomFactor(double factor);

    /// <summary>
    /// Sets max. zooming in.
    /// </summary>
    void ZoomTele();

    /// <summary>
    /// Sets min. zooming in.
    /// </summary>
    void ZoomWide();

    #endregion

    #region ----------------- SUPPLEMENTARY PART -----------------

    /// <summary>
    /// Turns the camera so that reference point is
    /// in the center of the frame area.
    /// </summary>
    /// <param name="point">Display point (in pixels) that is necessary to be put
    /// in the center by turning the camera.
    /// Starting point (0,0) is the upper-left corner of frame </param>
    /// <param name="frameSize">The frame size in pixels</param>
    void MoveTo(System.Drawing.Point point, System.Drawing.Size frameSize);

    /// <summary>
    /// turns and zooms the camera so that
    /// controlled rectangle takes a full frame area.
    /// If rectangle aspect ratio does not correspond to that of the frame,
    /// then zooming is executed so that the whole rectangle
       ///fits within the frame.
    /// The rectangle center fits the frame center.
    /// </summary>
    /// <param name="rect">Rectangle is set in pixels</param>
    /// <param name="frameSize">Frame size in pixels</param>
    void ShowRect(System.Drawing.Rectangle rect, System.Drawing.Size frameSize);

    #endregion
}

/// <summary>
/// Unified interface of camera input/output control implementation
/// </summary>
public interface IIOController
{
```

```
    /// <summary>
    /// Initialization
    /// </summary>
    /// <returns></returns>
    void Initialization();

   /// <summary>
   /// Returns the camera capabilities.
   /// </summary>
   /// <returns></returns>
   IOCapabilities GetCapabilities();

   /// <summary>
   /// Sets the defined value in the output
   /// </summary>
   /// <param name="portID">Output No.</param>
   /// <param name="value">1 or 0</param>
   void SetOutput(int portID, int value);

   /// <summary>
   /// Supplies pulse sequence (PWM) in indicated output.
   /// It is not supported by all cameras.
   /// </summary>
   /// <param name="portID">Output No.</param>
   /// <param name="pulses"> Pulse array </param>
   void SetOutput(int portID, IOPulse[] pulses)
}
```

Pan-and-tilt (IO controller) capabilities must be returned using **GetCapabilities()** method to inform the host about methods of optional capabilities implemented in the interface (if the device supports them).

DevType_RegInfo registration information of the device must be defined at the stage of loading the assembly and plugin in the class initialization method which realizes **IPlugin** interface by calling **RegisterDevType** host interface method (see Plugin registration in Eocortex).

Besides **DevType_RegInfo**, it is also necessary to complete the **RTFR_RegInfo** frame receiver information:

```
/// <summary>
/// The frame receiver registration information
/// </summary>
public class RTFR_RegInfo
{
    /// <summary>
    /// Device identifier
    /// </summary>
    public Guid DeviceTypeGuid;

       /// <summary>
       /// Data stream format
       /// </summary>
    public VideoStreamFormats StreamFormat;

       /// <summary>
       /// Connection protocol used
       /// </summary>
    public NetworkConnectionTypes ConnectionType;

       /// <summary>
       /// The device capabilities using the set format StreamFormat
       /// </summary>
```

```
        public RTFR_Capabilities Capabilities;
}
```

This information must be specified as many times as many different stream formats the IP device supports.

Similar to **DevType_RegInfo**, **RTFR_RegInfo** information must be defined at the stage of assembly and plugin loading in **IPlugin** interface of class initialization method. Call the host interface **RegisterDevType** method to complete it. Example of completing the classes is given below:

```
public void Initialize(IPluginHost host)
{
   DevType_RegInfo KingNet_KS3002MJ_Device = new DevType_RegInfo();
   KingNet_KS3002MJ_Device.DeviceTypeGuid =
         new Guid("5C49C51D-B17B-40b0-9656-6DC71DD86D90");
   KingNet_KS3002MJ_Device.DevTypeModelName = "KS3002MJ";
   KingNet_KS3002MJ_Device.DevTypeBrandName = "KingNet";
   KingNet_KS3002MJ_Device.AvailableResolutions = GetConcreteResolutions();
   KingNet_KS3002MJ_Device.Capabilities = DevType_Capabilities.SupportsCameras;
   KingNet_KS3002MJ_Device.GetPtzController = null;
   KingNet_KS3002MJ_Device.GetRTFR = (conParam, subParams) =>
   {
        RealTimeFrameReceiver rtfr = new RealTimeFrameReceiver(conParam,
   subParams);
        return rtfr;
   };
   KingNet_KS3002MJ_Device.SetDeviceParameters = (conParams, subParams) =>
   {
        return true;
   };

   host.RegisterDevType(KingNet_KS3002MJ_Device);

   RTFR_RegInfo rtfrKingNet_KS3002MJ_Mjpeg = new RTFR_RegInfo();
   rtfrKingNet_KS3002MJ_Mjpeg.DeviceTypeGuid =
         new Guid("5C49C51D-B17B-40b0-9656-6DC71DD86D90");
   rtfrKingNet_KS3002MJ_Mjpeg.StreamFormat = VideoStreamFormats.MJPEG;
   rtfrKingNet_KS3002MJ_Mjpeg.ConnectionType = NetworkConnectionTypes.UDP;
   rtfrKingNet_KS3002MJ_Mjpeg.Capabilities.SupportedStreamTypes =
         ChannelStreamTypes.MainVideo;
   rtfrKingNet_KS3002MJ_Mjpeg.Capabilities.SupportedExtraParameters =
       new DeviceParameters[] { DeviceParameters.Null, DeviceParameters.Null };
   rtfrKingNet_KS3002MJ_Mjpeg.Capabilities.SupportedDeviceParameters =
       new DeviceParameters[] { DeviceParameters.Null, DeviceParameters.Null };

   host.RegisterRTFR(rtfrKingNet_KS3002MJ_Mjpeg);
}
```