

[<Back](#)

Methods and events for creating drivers.

Updated: **August  
28, 2014**

# **C o n t e n t s**

•  
[1](#)  
[C](#)  
[o](#)  
[n](#)  
[s](#)  
[t](#)  
[a](#)  
[n](#)  
[t](#)  
[s](#)

•  
[2](#)  
[I](#)  
[R](#)  
[.](#)  
[C](#)  
[r](#)  
[e](#)  
[a](#)  
[t](#)  
[e](#)  
[D](#)  
[e](#)  
[v](#)  
[i](#)  
[c](#)  
[e](#)

-  
[2](#)  
[.](#)  
[1](#)  
[A](#)  
[V](#)  
[&](#)  
[o](#)

## Functions

<a href="#">IR.CreateDevice</a>	Creating a driver
<a href="#">Connect</a>	Connection to a device
<a href="#">Disconnect</a>	Disconnection from a device
<a href="#">IR.GetDevice</a>	Referring to a device
<a href="#">Set</a>	Setting up values in the device channel
<a href="#">Send</a>	Sending commands to a device
<a href="#">InvokeAction</a>	Sending commands to a UPNP device
<a href="#">Subscribe</a>	Subscribing to UPNP events
<a href="#">UnSubscribe</a>	Unsubscribing from UPNP events
<a href="#">HtmlDecode</a>	Substituting Html symbols
<a href="#">JSON.Stringify</a>	Converting a JSON object into a string
<a href="#">JSON.Parse</a>	Converting a string into a JSON object
<a href="#">new XML</a>	Creating XML objects
<a href="#">XML.ToString</a>	Converting an XML object into a string
<a href="#">SetFeedback</a>	Writing values in the feedback channel
<a href="#">GetFeedback</a>	Receiving values from the feedback channel
<a href="#">SetParameters</a>	Changing connection parameters
<a href="#">GetCommandAtName</a>	Referring to a command by its name or identifier
<a href="#">GetCommandAtPos</a>	Referring to a command by its position index
<a href="#">GetCommandsCount</a>	Receiving the quantity of commands
<a href="#">GetFeedbackAtName</a>	Receiving values from the feedback channel by their names or identifiers
<a href="#">GetFeedbackAtPos</a>	Receiving values from the feedback channel by their position index
<a href="#">GetFeedbacksCount</a>	Receiving the quantity of feedback channels
<a href="#">HexArrayToAsciiString</a>	Converting an array of Hex symbols into an ASCII string
<a href="#">IR.SendNotification</a>	Sending a local notification on iOS devices
<a href="#">IR.ClearNotify</a>	Clearing the list of notifications on iOS devices

## Events

<a href="#">EVENT_RECEIVE_DATA</a>	Receiving data from a device in the byte format
<a href="#">EVENT_RECEIVE_TEXT</a>	Receiving strings from a device
<a href="#">EVENT_RECEIVE_EVENT</a>	Receiving events from a device (UPNP Event)
<a href="#">EVENT_ONLINE</a>	Connection to a device is established
<a href="#">EVENT_OFFLINE</a>	Connection to a device is lost
<a href="#">EVENT_TAG_CHANGE</a>	Changing a tag value
<a href="#">EVENT_DEVICE_FOUND</a>	Finding UPnP devices
<a href="#">EVENT_CHANNEL_SET</a>	Activating of commands
<a href="#">EVENT_APP_ENTER_BACKGROUND</a>	Application goes into the background mode
<a href="#">EVENT_APP_ENTER_FOREGROUND</a>	Application goes into the foreground mode
<a href="#">EVENT_APP_WILL_TERMINATE</a>	Application finishes its work
<a href="#">EVENT_RECIEVE_NOTIFIEY</a>	Receiving local notifications

## Constants

```
// Send Mode IR.ALWAYS_CONNECTED = 0; // The driver is always connected, it can receive and
send data IR.CONNECT_WHEN_SENDING = 1; // The driver connects when sending data, it can
only send data
```

```
// Script Mode IR.DIRECT_AND_SCRIPT = 0; // Commands can be sent via scripts and editor
channels IR.SCRIPT_ONLY = 1; // Commands can be sent via scripts only
```

```
// Background Mode IR.BACKGROUND_OFF = 0; // The driver works only when the app is in the
foreground mode IR.BACKGROUND_ON = 1; // The driver works both when the app is in the
foreground and background modes
```

## IR.CreateDevice

This function is used for creating drivers.

## AV & Custom Systems TCP

### Example of creating DEVICE\_CUSTOM\_TCP

#### Syntax:

```
IR.CreateDevice(IR.DEVICE_CUSTOM_TCP, "Name", "Host", Port, SendMode, ScriptMode, BackgroundMode);
```

#### Parameters:

**IR.DEVICE\_CUSTOM\_TCP** (*driver type*) – the driver with the type of connection via TCP;  
**Name** (*string*) – the driver name;  
**Host** (*string*) – the device IP-address;  
**Port** (*number*) – the port on the device;  
**SendMode** (*mode*) – the mode of maintaining the connection session ([see constants](#));  
**ScriptMode** (*mode*) – the mode of command sending and communication with scripts ([see constants](#));  
**BackgroundMode** (*mode*) – the mode of driver work ([see constants](#)).

#### Example:

```
IR.CreateDevice(IR.DEVICE_CUSTOM_TCP, // Driver type
               "TCP 1", // Name
               "192.168.0.47", // IP-address
               123, // Port
               IR.ALWAYS_CONNECTED, // The driver always stays connected
               IR.SCRIPT_ONLY, // Commands can be sent via scripts
               IR.BACKGROUND_ON); // Background is on
```

```
IR.CreateDevice(IR.DEVICE_CUSTOM_TCP, // Driver type
               "TCP 1", // Name
               "192.168.0.47", // IP-address
               123, // Port
               IR.ALWAYS_CONNECTED, // The driver always stays connected
               IR.DIRECT_AND_SCRIPT, // Commands can be sent via scripts
               and channels
               IR.BACKGROUND_OFF); // Background is off
```

## AV & Custom UDP



## Example of creating DEVICE\_CUSTOM\_UDP

### Syntax:

```
IR.CreateDevice(IR.DEVICE_CUSTOM_UDP,  
               "Name",  
               "Host",  
               Port,  
               "Group",  
               Multicast,  
               LocalPort,  
               BackGroundMode);
```

### Parameters:

**IR.DEVICE\_CUSTOM\_UDP** (*driver type*) - the driver with the type of connection via UDP;  
**Name** (*string*) - the driver name;  
**Host** (*string*) - the device IP-address;  
**Port** (*number*) - the port on the device;  
**Group** (*string*) - null is Broadcast/ "IP" - IP multicast of the group;  
**Multicast** (*mode*) - the mode of sending the packet to the limited group of hosts;  
**LocalPort** (*number*) - the local port of the device;  
**Background\_mode** (*mode*) - the mode of driver work ([see constants](#)).

### Example:

```
IR.CreateDevice(IR.DEVICE_CUSTOM_UDP, // Driver type  
               "UDP 1", // Name  
               "192.168.0.47", // IP-address  
               6000, // Port  
               null,  
               false, // the packet is sent to all hosts in the network  
               0, // the external port is used  
               IR.BACKGROUND_ON); // Background is on
```

```
IR.CreateDevice(IR.DEVICE_CUSTOM_UDP, // Driver type  
               "UDP 2", // Name  
               "224.0.0.253", // IP-address  
               123, // Port  
               "224.0.0.253",  
               true, // the packet is sent to the group of hosts  
               0xFFFF, // any free is used  
               IR.BACKGROUND_OFF); // Background is off
```

```
IR.CreateDevice(IR.DEVICE_CUSTOM_UDP, // Driver type
```

```
"UDP 3", // Name
"192.168.0.47", // IP-address
123, // Port
null,
false, // the packet is sent to all hosts in the network
1033, // port 1033 is used
IR.BACKGROUND_ON); // Background is on
```

## AV & Custom HTTP



### Example of creating DEVICE\_CUSTOM\_HTTP\_TCP

#### Syntax:

```
IR.CreateDevice(IR.DEVICE_CUSTOM_HTTP_TCP,
    "Name",
    "Host",
    Port,
    "Login",
    "Password",
    SSL,
    BackGroundMode);
```

#### Parameters:

**IR.DEVICE\_CUSTOM\_HTTP\_TCP** (*driver type*) - the driver with the type of connection via HTTP/TCP;  
**Name** (*string*) - the driver name;  
**Host** (*string*) - the device IP-address;  
**Port** (*number*) - the port on the device;  
**Login** (*string*) - the login when using the secure session;  
**Password** (*string*) - the password when using the secure session;  
**SSL** (*true/false*) - the security protocol for protecting confidential network data;  
**BackgroundMode** (*mode*) - the mode of driver work ([see constants](#)).

#### Example:

```
IR.CreateDevice(IR.DEVICE_CUSTOM_HTTP_TCP, // Driver type
    "HTTP 1", // Name
    "weather.yahooapis.com", // web address
    80, // Port
    "admin", // Login
    "123", // Password
    true, // SSL is ON
    IR.BACKGROUND_ON); // Background is on
```

```
IR.CreateDevice(IR.DEVICE_CUSTOM_HTTP_TCP, // Driver type
               "HTTP 2", // Name
               "192.168.0.1", // IP-address
               23, // Port
               "root", // Login
               "atata", // Password
               false, // SSL is OFF
               IR.BACKGROUND_OFF); // Background is off
```

## Custom Server TCP



### Example of creating DEVICE\_CUSTOM\_SERVER\_TCP

#### Syntax:

```
IR.CreateDevice(IR.DEVICE_CUSTOM_SERVER_TCP, "Name", "Host", Port,
               MaxClients, BackgroundMode);
```

#### Parameters:

**IR.DEVICE\_CUSTOM\_SERVER\_TCP** (*driver type*) - the driver with the type of connection via TCP;

**Name** (*string*) - the driver name;

**Host** (*string*) - the device IP-address;

**Port** (*number*) - the port on the device;

**MaxClients** (*number*) - the maximum number of clients for connection;

**BackgroundMode** (*mode*) - the mode of driver work ([see constants](#)).

#### Example:

```
IR.CreateDevice(IR.DEVICE_CUSTOM_SERVER_TCP, // Driver type
               "SERVER TCP 1", // Name
               "192.168.0.1", // IP-address
               2323, // Port
               4, // four clients can be connected at a time
               IR.BACKGROUND_ON); // Background is on
```

```
IR.CreateDevice(IR.DEVICE_CUSTOM_SERVER_TCP, // Driver type
               "SERVER TCP 2", // Name
               "192.168.0.1", // IP-address
               23, // Port
               5, // five clients can be connected at a time
```

```
IR.BACKGROUND_OFF); // Background is off
```

## Custom Server UDP



Example of creating DEVICE\_CUSTOM\_SERVER\_UDP

### Syntax:

```
IR.CreateDevice(IR.DEVICE_CUSTOM_SERVER_UDP,  
               "Name",  
               "Host",  
               Port,  
               MaxClients,  
               ClientTimeOut,  
               BackGroundMode);
```

### Parameters:

**IR.DEVICE\_CUSTOM\_SERVER\_UDP** (*driver type*) - the driver with the type of connection via UDP;

**Name** (*string*) - the driver name;

**Host** (*string*) - the device IP-address;

**Port** (*number*) - the port on the device;

**MaxClients** (*number*) - the maximum number of clients for connection;

**ClientTimeOut** (*number*) - the time when the server is waiting for the incoming data; if the time runs out, the client will be disconnected (ms);

**BackgroundMode** (*mode*) - the mode of driver work ([see constants](#)).

### Example:

```
IR.CreateDevice(IR.DEVICE_CUSTOM_SERVER_UDP // Driver type  
               "SERVER UDP 1", // Name  
               "192.168.0.100", // IP-address  
               2323, // Port  
               4, // four clients can be connected at a time  
               100, // the server is waiting for the incoming data for 100  
ms  
               IR.BACKGROUND_ON); // Background is on
```

```
IR.CreateDevice(IR.DEVICE_CUSTOM_SERVER_UDP, // Driver type  
               "SERVER UDP 2", // Name  
               "192.168.0.100", // IP-address
```

```
23, // Port
5, // five clients can be connected at a time
100, // the server is waiting for the incoming data for 100
ms
IR.BACKGROUND_OFF); // Background is off
```

## AMX

### Example of creating DEVICE\_AMX

#### Syntax:

```
IR.CreateDevice(IR.DEVICE_AMX,
    "Name",
    "Host",
    Port,
    PanelID,
    "Login",
    "Password",
    BackGroundMode);
```

#### Parameters:

**IR.DEVICE\_AMX** (*driver type*) - the driver with the type of connection via TCP;

**Name** (*string*) - the driver name;

**Host** (*string*) - the device IP-address;

**Port** (*number*) - the port on the device;

**PanelID** (*number*) - the panel identifier, it has to be unique;

**Login** (*string*) - the login when using the secure session;

**Password** (*string*) - the password when using the secure session;

**BackgroundMode** (*mode*) - the mode of driver work ([see constants](#)).

#### Example:

```
IR.CreateDevice(IR.DEVICE_AMX, // Driver type
    "AMX 1", // Name
    "192.168.0.150", // IP-address
    1319, // Port
    12, // the panel identifier
    "admin", // Login
    "123", // Password
```

```
IR.BACKGROUND_ON); // Background is on
```

```
IR.CreateDevice(IR.DEVICE_AMX, // Driver type  
               "AMX 2", // Name  
               "192.168.0.150", // IP-address  
               23, // Port  
               13, // the panel identifier  
               "root", // Login  
               "atata", // Password  
               IR.BACKGROUND_OFF); // Background is off
```

## KNX

### Example of creating DEVICE\_EIB\_UDP

#### Syntax:

```
IR.CreateDevice(IR.DEVICE_EIB_UDP,  
               "Name",  
               "Host",  
               Port,  
               ConnectionWaitTime,  
               SendWaitTime,  
               PingTime,  
               BackgroundMode);
```

#### Parameters:

**IR.DEVICE\_EIB\_UDP** (*driver type*) - the driver with the type of connection via UDP;

**Name** (*string*) - the driver name;

**Host** (*string*) - the device IP-address;

**Port** (*number*) - the port on the device;

**ConnectionWaitTime** (*number*) - the time of waiting at emergency disconnection (ms);

**SendWaitTime** (*number*) - the pause between command sending to the bus, to decrease the load of the router and the bus (ms);

**PingTime** (*number*) - frequency of sending the Ping command to check connection with the router (ms);

**BackgroundMode** (*mode*) - the mode of driver work ([see constants](#)).

#### Example:

```

IR.CreateDevice(IR.IR.DEVICE_EIB_UDP, // Driver type
               "EIB 1", // Name
               "192.168.0.150", // IP-address
               71, // Port
               500, // at emergency disconnection the waiting time is 500
ms
               500, // commands are sent each 500 ms
               500, // the connection with the router is checked each 500
ms
               IR.BACKGROUND_ON); // Background is on

IR.CreateDevice(IR.IR.DEVICE_EIB_UDP, // Driver type
               "EIB 2", // Name
               "192.168.0.150", // IP-address
               72, // Port
               500, // at emergency disconnection the waiting time is 500
ms
               500, // commands are sent each 500 ms
               500, // the connection with the router is checked each 500
ms
               IR.BACKGROUND_OFF); // Background is off

```

## BAOS 1(770)



### Example of creating DEVICE\_BAOS\_1

#### Syntax:

```
IR.CreateDevice(IR.DEVICE_BAOS_1, "Name", "Host", Port, PingTime,
BackgroundMode);
```

**Parameters:** **IR.DEVICE\_BAOS\_1** (*driver type*) - the driver with the type of connection via TCP;  
**Name** (*string*) - the driver name;

**Host** (*string*) - the device IP-address;

**Port** (*number*) - the port on the device;

**PingTime** (*number*) - frequency of sending the Ping command to check connection with the router (ms);

**BackgroundMode** (*mode*) - the mode of driver work ([see constants](#)).

#### Example:

```
IR.CreateDevice(IR.DEVICE_BAOS_1, // Driver type
```

```

        "BAOS_1_1", // Name
        "192.168.0.47", // IP-address
        71, // Port
        500, // the connection with the router is checked each 500
ms (not less than 500 is recommended)
        IR.BACKGROUND_ON); // Background is on

IR.CreateDevice(IR.DEVICE_BAOS_1, // Driver type
        "BAOS_1_2", // Name
        "192.168.0.47", // IP-address
        23, // Port
        500, // the connection with the router is checked each 500
ms (not less than 500 is recommended)
        IR.BACKGROUND_OFF); // Background is off

```

## BAOS 2(771/772)



### Example of creating DEVICE\_BAOS\_2

#### Syntax:

```

IR.CreateDevice(IR.DEVICE_BAOS_2, "Name", "Host", Port, PingTime,
BackgroundMode);

```

#### Syntax:

**IR.DEVICE\_BAOS\_2** (*driver type*) - the driver with the type of connection via TCP;

**Name** (*string*) - the driver name;

**Host** (*string*) - the device IP-address;

**Port** (*number*) - the port on the device;

**PingTime** (*number*) - frequency of sending the Ping command to check connection with the router (ms);

**BackgroundMode** (*mode*) - the mode of driver work ([see constants](#)).

#### Example:

```

IR.CreateDevice(IR.DEVICE_BAOS_2, // Driver type
        "BAOS_2_1", // Name
        "192.168.0.47", // IP-address
        71, // Port
        500, // the connection with the router is checked each 500
ms (not less than 500 is recommended)

```

```

        IR.BACKGROUND_ON); // Background is on

IR.CreateDevice(IR.DEVICE_BAOS_2, // Driver type
               "BAOS_2_2", // Name
               "192.168.0.47", // IP-address
               72, // Port
               500, // the connection with the router is checked each 500
ms (not less than 500 is recommended)
               IR.BACKGROUND_OFF); // Background is off

```

## CRESTRON

### Example of creating DEVICE\_CRESTRON

#### Syntax:

```

IR.CreateDevice(IR.DEVICE_CRESTRON,
               "Name",
               "Host",
               Port,
               NetID,
               "TelnetLogin",
               "TelnetPassword",
               "TelnetPort",
               SSL,
               BackGroundMode);

```

#### Parameters:

**IR.DEVICE\_CRESTRON** (*driver type*) - the driver with the type of connection via TCP;

**Name** (*string*) - the driver name;

**Host** (*string*) - the device IP-address;

**Port** (*number*) - the port on the device;

**NetID** (*число*) - the panel identifier in the SIMPL project (decimal number);

**TelnetLogin** (*string*) - the login when using the secure session;

**TelnetPassword** (*string*) - the password when using the secure session;

**TelnetPort** (*string*) - the port for activated (41797)/deactivated (41795) SSL;

**SSL** (*true/false*) - the security protocol for protecting confidential network data;

**BackgroundMode** (*mode*) - the mode of driver work ([see constants](#)).

### Example:

```
IR.CreateDevice(IR.DEVICE_CRESTRON, // Driver type
               "CRESTRON 1", // Name
               "192.168.0.47", // IP-address
               71, // Port
               12, // the panel identifier
               "admin", // Login
               "123", // Password
               "41797", // the port of the activated SSL
               true, // SSL is ON
               IR.BACKGROUND_ON); // Background is on
```

```
IR.CreateDevice(IR.DEVICE_CRESTRON, // Driver type
               "CRESTRON 2", // Name
               "192.168.0.47", // IP-address
               72, // Port
               12, // the panel identifier
               "admin", // Login
               "123", // Password
               "41795", // the port of the deactivated SSL
               false, // SSL is OFF
               IR.BACKGROUND_OFF); // Background is off
```

## IR.GetDevice

This function is used for accessing the driver.

### Syntax:

```
IR.GetDevice("Name");
```

### Parameters:

**Name** (string) – the driver name from Project Device Panel.  
For HDL drivers the network name (**NetworkName**) is used instead of the driver name.

### Example:

```
var MyDriver = IR.GetDevice("Dune HD"); // Receive access to the Dune HD driver
```

```
var MyDriverHDL = IR.GetDevice("HDL-BUS Pro Network (RS232)"); // Receive access to RS232 of the HDL network
```

## Send

This function is used for sending data to devices.

**Syntax 1:** Sending of multiple variables instructions to your device:

```
IR.GetDevice("Device_Name").Send([command_1, .. , command_n]);
```

- **Device\_Name** - the name of the device created in iRidium GUI or iRidiumScript
- **command\_1** - the first variable or string - instruction that is sent to the device
- **command\_n** - the last variable or string - instruction that is sent to the device

**Syntax 2:** Sending an array of instructions to your device:

```
IR.GetDevice("Device_Name").Send(array_command);
```

- **Device\_Name** - the name of the device created in iRidium GUI or iRidiumScript
- **array\_command** - an array of instructions to be sent to the device

```
IR.AddListener(IR.EVENT_START, 0, function()  
// Event is activated at the application launch  
{  
  IR.CreateDevice(IR.DEVICE_CUSTOM_TCP, "DEVICE", "192.168.0.116", 80);  
//Command for creating drivers  
  IR.GetDevice("DEVICE").Send(['getverion\r\n HTTP 1.1', '\r\n\r\n']);  
// Command sends instructions to the device ('getverion\r\n HTTP  
1.1', '\r\n\r\n')  
});
```

If you use the HTTP driver, then add "**GET**," "**PUT**," "**POST**," in the beginning of the string:

```
IR.AddListener(IR.EVENT_START, 0, function()  
{  
  IR.GetDevice("DEVICE").Send(['GET,getverion\r\n HTTP 1.1', '\r\n\r\n']);  
  
  IR.GetDevice("DEVICE").Send(['PUT,getverion\r\n HTTP 1.1', '\r\n\r\n']);  
  
  IR.GetDevice("DEVICE").Send(['POST,getverion\r\n HTTP 1.1', '\r\n\r\n']);  
});
```

## Connect

This function establishes connection with devices.

**Syntax:** IR.GetDevice('Device\_Name').Connect;

- **Device\_Name** - the name of the device created in iRidium GUI or iRidiumScript

```
IR.GetDevice("Global cache").Connect; //Establish connection with the device
```

# Disconnect

This function breaks connection with devices.

**Syntax:** IR.GetDevice('Device\_Name').Disconnect;

- **Device\_Name** - the name of the device created in iRidium GUI or iRidiumScript

```
IR.GetDevice("Global cache").Disconnect; //Break connection with the device
```

# Set

This function is used for activating command channels and setting values in them. The value setting is possible for native drivers only.

**Syntax:** IR.GetDevice('Device\_Name').Set(channel,value);

- **Device\_Name** - the name of the device created in Project Device Panel or iRidiumScript
- **channel** - the channel identifier (name or number)
- **value** - the value written in the channel

```
// Set the red value for the channel with number 1 (its number in the list of driver commands)
```

```
IR.GetDevice("KNX Router (KNXnet/IP)").Set(1,"red");
```

```
// Set the value (100) and activate the channel with the name BAOS Command
```

```
IR.GetDevice("BAOS").Set("BAOS Command",100);
```

```
// Send the command POWER ON from the IR output of Global Cache
```

```
IR.GetDevice("Global Cache").Set("POWER ON","");
```

```
// For script drivers send an empty value "".
```

```
IR.GetDevice("Sonos").Set("Play","");
```

Syntax used for **HDL-BUS Pro** and **Domintell**:

```
IR.GetDevice("NetWork").Set("Device_Name:channel",value);
```

- **NetWork** - the name of the HDL network, it can be seen in Project Device Panel
- **Device\_Name** - the name of the device created in Project Device Panel
- **channel** - the channel identifier (name or number)
- **value** - the value written in the channel

```
// Set up value 0 and activate the channel Read On Start of the sub-device Sensor_7
```

```
IR.GetDevice("HDL-BUS Pro Network (UDP)").Set("Relay_1:Channel_3",0);
```

```
// Set up value 1 and activate the channel Command_On of the sub-device Domintell
```

```
IR.GetDevice("Domintell Network (UDP)").Set("Domintell Device:Command_On",
```

1);

## InvokeAction

This function is used for sending instructions to devices with UPnP support

**Syntax:** IR.GetDevice('Device\_Name').InvokeAction(ActionName,ServiceType,Arguments);

- **Device\_Name** - the name of the device created in iRidium GUI or iRidiumScript
- **ActionName** - the command name
- **ServiceType** - the service used
- **Arguments** - the list of command arguments

```
IR.GetDevice("BAOS").InvokeAction("Play", "service:AVTransport:1",  
{InstanceID: 0, Speed: 1} ); //Sending the command to the UPnP device
```

## Subscribe

This function is used for subscribing to events of devices with UPnP support.

**Syntax:** IR.GetDevice('Device\_Name').Subscribe(ServiceType);

- **Device\_Name** - the name of the device created in iRidium GUI or iRidiumScript
- **ServiceType** - the service used

```
IR.GetDevice("BAOS").Subscribe("urn:schemas-upnp-org:service:AVTransport:1");  
//Subscribing to UPnP events
```

## UnSubscribe

This function is used for unsubscribing to events of devices with UPnP support.

**Syntax:** IR.GetDevice('Device\_Name').UnSubscribe(ServiceType);

- **Device\_Name** - the name of the device created in iRidium GUI or iRidiumScript
- **ServiceType** - the service used

```
IR.GetDevice("BAOS").Unsubscribe("urn:schemas-upnp-  
org:service:AVTransport:1");  
// Unsubscribing from UPnP events
```

## HtmlDecode

This function is used for substituting html symbols.

**Syntax:** IR.HtmlDecode(text)

- **text** - the text with incorrect coding

```
IR.AddListener(IR.EVENT_RECEIVE_TEXT, IR.GetDevice("Global Cache"),
function(text)
// Event is activated when receiving data from the device
{
  IR.HtmlDecode(text);//Transcode html tags
});
```

## JSON.Stringify

This function converts JSON objects to strings.

**Syntax:** JSON.Stringify(Object)

- **Object** - the JSON object

```
var text = JSON.Stringify(Object); //Convert the JSON object to the string;
```

## JSON.Parse

This function converts strings to JSON objects.

**Syntax:** JSON.Parse(Text)

- **Text** - the string

```
var JSONObject = JSON.Parse (text); //Convert the string to the JSON object
```

## New XML

This function is used for creating XML objects

**Syntax:** var xml = new Xml(Text);

- **Text** - the string containing the xml message

```
var xml = new Xml(Text); //Create the XML string
```

## XML.ToString

This function is used for converting XML objects to strings.

**Syntax:** var str = xml.ToString();

- **xml** - the XML object

```
var str = xml.toXMLString();//Convert the xml object to the string
```

## SetFeedback

This function is used for writing values (a string or number) in a feedback channel. The feedback channel should be created in Project Device Panel. It is used for displaying data received from script drivers.

**Syntax:** IR.GetDevice("MyDevice").SetFeedback("MyFeedbackChannel", value);

- **MyDevice** - the name of the device from Project Device Panel
- **MyFeedbackChannel** - the name of the feedback channel from Project Device Panel
- **value** - the value (string or number) you want to write in the channel

```
IR.AddListener(IR.EVENT_START,0,function()  
{  
    IR.GetDevice("MyDevice").SetFeedback("Room", "Kitchen");  
});
```

## GetFeedback

This function is used for receiving values from feedback channels. The feedback channels should be created in Project Device Panel.

**Syntax:** IR.GetDevice("MyDevice").GetFeedback("MyFeedbackChannel");

- **MyDevice** - the device name from Project Device Panel
- **MyFeedbackChannel** - the feedback channel name from Project Device Panel

```
IR.AddListener(IR.EVENT_START,0,function()  
{  
    // Output the value of the feedback channel Room  
    // of the MyDevice driver in the log  
  
    IR.Log(IR.GetDevice("MyDevice").GetFeedback("Room"));  
});
```

## SetParameters

This function is used for changing parameters of connection of the driver to the controlled device. A typical example of switching LAN - WAN networks.

The SetParameters function **does not physically rewrite** the driver connection settings in projects, so new parameters are not saved after closing the application.

**Syntax:** IR.GetDevice("MyDevice").SetParameters({Host: NewHost, Port: NewPort});

- **MyDevice** - the name of the device from Project Device Panel or created dynamically
- **NewHost** - the IP-address of the device you need to connect to
- **NewPort** - the port for connection to the device

```
IR.AddListener(IR.EVENT_START,0,function()
{
    IR.GetDevice("MyDevice").SetParameters({Host: "192.168.0.21", Port:
"9090"});
});
```

If the driver has additional specific parameters, for example **KNX BAOS 771/772 (TCP)** has the **UpdateTime** parameter besides **Host** and **Port**, additional parameters should be indicated.

```
IR.AddListener(IR.EVENT_START,0,function()
{
    IR.GetDevice("BAOS").SetParameters({Host: "192.168.0.163", Port: "12004",
UpdateTime: "0"});
});
```

**AMX** has the **DeviceID**, **Login**, **Password** parameters besides **Host** and **Port**. Additional parameters should be indicated.

```
IR.AddListener(IR.EVENT_START,0,function()
{
    IR.GetDevice("AMX").SetParameters({
        Host: "192.168.0.21",
        Port: "12004",
        DeviceID: "1",
        Login: "admin",
        Password: "12345"});
});
```

**AV & Custom Systems (HTTP)** has the **SSL**, **Login**, **Password** parameters besides **Host** and **Port**. Additional parameters should be indicated.

```
IR.AddListener(IR.EVENT_START,0,function()
{
    IR.GetDevice("AV & Custom Systems (HTTP)").SetParameters({
        Host: "192.168.0.21",
        Port: "12004",
        SSL: "1",
        Login: "admin",
        Password: "12345"});
});
```

#### **AV & Custom Systems (UDP)**

```
IR.AddListener(IR.EVENT_START,0,function()
{
    IR.GetDevice("AV & Custom Systems (UDP)").SetParameters({
        Host: "192.168.0.21",
        Port: "2500",
```

```
        ScriptMode: 0,  
        LocalPort: "2500",  
        BackGroundMode: 0});  
});
```

### **AV & Custom Systems (TCP)**

```
IR.AddListener(IR.EVENT_START,0,function()  
{  
    IR.GetDevice("AV & Custom Systems (UDP)").SetParameters({  
        Host: "192.168.0.21",  
        Port: "8080",  
        ScriptMode: 0,  
        BackGroundMode: 0});  
});
```

**Crestron** has the **NetID** parameter besides **Host** and **Port**. Additional parameters should be indicated.

```
IR.AddListener(IR.EVENT_START,0,function()  
{  
    IR.GetDevice("Crestron").SetParameters({  
        Host: "192.168.0.21",  
        Port: "12004",  
        NetID: "1"});  
});
```

**DuoTecno** does not have additional parameters. The parameters Host and Port should be indicated.

```
IR.AddListener(IR.EVENT_START,0,function()  
{  
    IR.GetDevice("DuoTecno").SetParameters({  
        Host:"192.168.0.21",  
        Port:"5001" });  
});
```

**Modbus(TCP)** has the **UpdateTime** parameter besides **Host** and **Port**. Additional parameters should be indicated.

```
IR.AddListener(IR.EVENT_START,0,function()  
{  
    IR.GetDevice("Modbus").SetParameters({  
        Host: "192.168.0.21",  
        Port: "502",  
        UpdateTime: "1500"});  
});
```

**UPnP** has the **DescriptionUrl** parameter besides **Host** and **Port**. Additional parameters should be indicated.

```
IR.AddListener(IR.EVENT_START,0,function()  
{  
    IR.GetDevice("UPnP").SetParameters({  
        DescriptionUrl: "wiki2.iridiummobile.net"  
        Host: "192.168.0.21",  
        Port: "8080"});  
});
```

**KNX IP Router (KNXnet/IP)** has the **ConnectTime**, **SendTime**, **PingTime**, **Nat** parameters besides **Host** and **Port**. Additional parameters should be indicated.

```
IR.AddListener(IR.EVENT_START,0,function()  
{  
    IR.GetDevice('KNX IP Router').SetParameters({  
        Host: '217.115.10.10',  
        Port: '3671',  
        ConnectTime: '120000',  
        SendTime: '0',  
        PingTime: '60000',  
        Nat: '1'});  
});
```

The Nat parameter is a switcher: 0 - to turn off, 1 - to turn on. It turns on/off connection via the NAT server. It is required for work via UDP over the Internet. It is 0 by default.

Change of LAN - WAN connections for **HDL**, script, [download](#)

## GetCommandAtName

The function enables reference to commands of the AV & Custom System driver (HTTP, TCP, UDP, RS232) by their names or identifiers and receiving data from the Data field.

**Syntax:** IR.GetDevice(DriverName).GetCommandAtName(CommandName);

- **DriverName (variable or string)** - the device name from Project Device Panel
- **CommandName (variable or string)** - the command name/identifier (begins with 1)

```
// by the name  
var NameCommand = IR.GetDevice("MyDevice").GetCommandAtName("Command 1");  
IR.Log("Name: NameCommand = id:" + NameCommand .id + " name:" +  
NameCommand.name + " data:" + NameCommand.data);  
// by the identifier  
var NameCommand = IR.GetDevice("MyDevice").GetCommandAtName(1);  
IR.Log("ID: NameCommand = id:" + NameCommand .id + " name:" +  
NameCommand.name + " data:" + NameCommand.data);
```

### Output data:

- id: the command identifier
- data: the command hex data
- name: the command name

## GetCommandAtPos

The function enables reference to commands of the AV & Custom System driver (HTTP, TCP, UDP, RS232) by their position index and receiving data from the Data field.

**Syntax:** IR.GetDevice(DriverName).GetCommandAtPos(CommandPos);

- **DriverName (variable or string)** - the device name from Project Device Panel
- **CommandPos (whole number)** - the command position index (begins with 0)

```
// by position index
var PosCommand = IR.GetDevice("MyDevice").GetCommandAtPos(0);
IR.Log("POS: PosCommand = id:" + PosCommand.id + " name:" + PosCommand.name +
" data:" + PosCommand.data);
```

### Output data:

- id: the command identifier
- data: the command hex data
- name: the command name

## GetCommandsCount

The function enables reference to the AV & Custom System driver (HTTP, TCP, UDP, RS232) and receiving the quantity of commands in projects.

**Syntax:** IR.GetDevice(DriverName).GetCommandsCount();

- **DriverName (variable or string)** - the device name from Project Device Panel

```
// receive quantity
var CommandsCount = IR.GetDevice("MyDevice").GetCommandsCount();
// output the command quantity in the log
IR.Log("CommandsCount = " + CommandsCount);
```

## GetFeedbackAtName

The function enables reference to feedback channels of the AV & Custom System driver (HTTP, TCP, UDP, RS232) by their names or identifiers and receiving data from the Data field.

**Syntax:** IR.GetDevice(DriverName).GetFeedbackAtName(FeedbackName);

- **DriverName (variable or string)** - the device name from Project Device Panel
- **FeedbackName (variable or string)** - the feedback channel name/identifier (begins with 1)

```

// by the name
var NameFeedback = IR.GetDevice("MyDevice").GetFeedbackAtName("Feedback
1");
IR.Log("Name: NameFeedback = id:" + NameFeedback.id + " name:" +
NameFeedback.name + " data:" + NameFeedback.data);
// by the identifier
var NameFeedback = IR.GetDevice("MyDevice").GetFeedbackAtName("Feedback
1");
IR.Log("ID: NameFeedback = id:" + NameFeedback.id + " name:" +
NameFeedback.name + " data:" + NameFeedback.data);

```

### Output data:

- id: the feedback channel identifier
- data: the feedback channel hex data
- name: the feedback channel name

## GetFeedbackAtPos

The function enables reference to feedback channels of the AV & Custom System driver (HTTP, TCP, UDP, RS232) by their position index and receiving data from the Data field.

**Syntax:** IR.GetDevice(DriverName).GetFeedbackAtPos(FeedbackPos);

- **DriverName (variable or string)** - the device name from Project Device Panel
- **FeedbackPos (whole number)** - the feedback channel position index (begins with 0)

```

// by position index
var PosFeedback = IR.GetDevice("MyDevice").GetFeedbackAtPos(0);
IR.Log("POS: PosFeedback= id:" + PosFeedback.id + " name:" + PosFeedback.name
+ " data:" + PosFeedback.data);

```

### Output data:

- id: the feedback channel identifier
- data: the feedback channel hex data
- name: the feedback channel name

## GetFeedbacksCount

The function enables reference to the AV & Custom System driver (HTTP, TCP, UDP, RS232) and receiving the quantity of feedback channels in projects.

**Syntax:** IR.GetDevice(DriverName).GetFeedbacksCount();

- **DriverName ((variable or string)** - the device name from Project Device Panel

```

// receive quantity
var FeedbacksCount = IR.GetDevice("MyDevice").GetFeedbacksCount();
// output the quantity of feedback channels in the log

```

```
IR.Log("FeedbacksCount = " + FeedbacksCount);
```

## HexArrayToAsciiString

The function enables conversion of an array of Hex symbols into an ASCII string.

**Syntax:** var l\_sStr += String.fromCharCode(parseInt(in\_aArray["Index"], "N"));

- **Index** - the index of the symbol in the Hex array
- **N** - a number between 2 and 36 designating the numeral system

With the help of the parameter of the numeral system you can indicate in to what numeral system the number in the string belongs. When the method is executed, the number will be converted in the decimal numeral system.

```
// conversion of the array of Hex symbols into an ASCII string
function HexArrayToAsciiString(in_aArray)
{
    // initialization of the string
    var l_sStr = '';

    // traversing the array
    for(var i = 0; i < in_aArray.length; i++)
    // converting the array into the string
        l_sStr += String.fromCharCode(parseInt(in_aArray[i], 16));
    // вернем строку
    return l_sStr;
}
```

```
IR.AddListener(IR.EVENT_START, 0, function()
{
    // receive by the position index
    var MyCommand = IR.GetDevice("MyDevice").GetCommandAtPos(0); // 0 –
the command position index
    IR.Log("MyCommandData = "+HexArrayToAsciiString(MyCommand.data));
    var MyFeedback = IR.GetDevice("MyDevice").GetFeedbackAtPos(0); // 0 –
the feedback channel position index
    IR.Log("MyFeedbackData = "+HexArrayToAsciiString(MyFeedback.data));
    //receive by the name
    var MyCommand = IR.GetDevice("MyDevice").GetCommandAtName("Command 1");
    IR.Log("MyCommandData = " + HexArrayToAsciiString(MyCommand.data));
    var MyFeedback = IR.GetDevice("MyDevice").GetFeedbackAtName("Feedback
1");
    IR.Log("MyFeedbackData = " + HexArrayToAsciiString(MyFeedback.data));
});
```

## IR.SendNotification

The function is used for sending local notifications on iOS devices.

**Syntax:** IR.SendNotification(text, delay, sound\_id, badge\_increment\_num, id);

- **text** - the notification text
- **delay** (default=0) - the delay of showing the notification (in seconds)
- **sound\_id** (default=0) - the sound identifier which will accompany the notification
- **badge\_increment\_num** (default=0) - by what number the number on the notification icon on the app icon has to be increased
- **id** (default=None) - the unique notification identifier

</pre> Example:

```
        // Sound identifiers
        SOUND_ID = {
            None : 0, // without sound
            Default : 1, // standard notification sound on iOS
            Ringing : 2, // standard ringtone
        };
        // Creation of required variables
        string = "Crestron " + name + " value = " + value;
        id = "" + name;
        delay = 0; // seconds
        sound_id = SOUND_ID.Default;
        // badge - the numeric icon on the app icon
        badge_increment_num = 1;
        IR.SendNotification(string, delay, sound_id, badge_increment_num, id);
    });
```

## IR.ClearNotification

The function is used for clearing the list of notifications on iOS devices.

Example:

```
//when clicking on the button with the name "clr_notty" on Page 1 clear the
list of notifications on the iOS device
    clr_notty = IR.GetPage("Page 1").GetItem("clr_notty");
    IR.AddListener(IR.EVENT_ITEM_PRESS, clr_notty, function() {
        IR.ClearNotification();
    });
```

[↑ Back](#)

## EVENT\_RECEIVE\_DATA

The event is activated when receiving data in the byte format from devices.

**Syntax:** IR.AddListener(IR.EVENT\_RECEIVE\_DATA, IR.GetDevice(Device\_Name), function(text)

- **Device\_Name** - the name of the device created in iRidium GUI or iRidiumScript

```
IR.AddListener(IR.EVENT_RECEIVE_DATA, IR.GetDevice("Global Cache"),
function(text)
//Event is activated when receiving data from the device
{
IR.Log(text) //Output the text in the log
});
```

## EVENT\_RECEIVE\_TEXT

The event is activated when receiving data in the string format from devices.

**Syntax:** IR.AddListener(IR.EVENT\_RECEIVE\_TEXT, IR.GetDevice(Device\_Name), function(text)

- **Device\_Name** - the name of the device created in iRidium GUI or iRidiumScript

```
IR.AddListener(IR.EVENT_RECEIVE_TEXT, IR.GetDevice("Global Cache"),
function(text)
// The event is activated when receiving data from the device
{
IR.Log(text) // Output the text in the log
});
```

## EVENT\_ONLINE

The event is activated when establishing connection with devices.

**Syntax:** IR.AddListener(IR.EVENT\_ONLINE , IR.GetDevice(Device\_Name), function()

- **Device\_Name** - the name of the device created in iRidium GUI or iRidiumScript

```
IR.AddListener(IR.EVENT_ONLINE , IR.GetDevice("Global Cache"), function()
// The event is activated when receiving events from the device
{
IR.Log("Device is online") // Output the text in the log
});
```

## EVENT\_OFFLINE

The event is activated when breaking connection with devices.

**Syntax:** IR.AddListener(IR.EVENT\_OFFLINE , IR.GetDevice(Device\_Name), function()

- **Device\_Name** - the name of the device created in iRidium GUI or iRidiumScript

```

IR.AddListener(IR.EVENT_OFFLINE , IR.GetDevice("Global Cache"), function()
// The event is activated when receiving events from the device
{
IR.Log("Device is offline") // Output the text in the log
});

```

## EVENT\_TAG\_CHANGE

The event is activated when changing tags on devices (for native drivers only).

**Syntax:** IR.AddListener(IR.EVENT\_TAG\_CHANGE, IR.GetDevice(Device\_Name), function(name, value)

- **Device\_Name** - the name of a native driver in Project Device Panel
- **name** - the name of the changed tag
- **value** - the new tag value

```

IR.AddListener(IR.EVENT_TAG_CHANGE, IR.GetDevice("KNX"), function(name,
value)
// The event is activated when receiving events from the device
{
IR.Log(name,value) // Output the tag name and new value in the log
});

```

**Syntax for HDL:** IR.AddListener(IR.EVENT\_TAG\_CHANGE, IR.GetDevice(Network\_Name), function(name, value)

- **Network\_Name** - the name of the HDL network in Project Device Panel
- **name** - the variable receiving the device name + colon + the name of the changed channel
- **value** - the new channel value

```

IR.AddListener(IR.EVENT_TAG_CHANGE, IR.GetDevice("HDL-BUS Pro Network
(UDP)", function(name, value)
// The event is activated when receiving events from the device
{
// If it is required to check the channel value,
// it should be done as follows:
if (name == "Logic:BMS Enable")
{
// Logic – the device name
// BMS Enable – the channel name
IR.Log(name,value) // Output the tag name and new value in the log
}
});

```

## EVENT\_DEVICE\_FOUND

The event is activated when iRidium finds UPnP devices.

**Syntax:** IR.AddListener(IR.EVENT\_DEVICE\_FOUND, 0, function(name)

- **Device\_Name** - the name of the device created in iRidium GUI or iRidiumScript \* **name** - the name of the found device

```
IR.AddListener(IR.EVENT_DEVICE_FOUND, 0, function(name)
// The event is activated when finding UPnP devices
{
IR.Log(name) // Output the name of the found device in the log
});
```

## EVENT\_RECEIVE\_EVENT

The event is activated when some event is activated on a UPnP device.

**Syntax:** IR.AddListener(IR.EVENT\_RECEIVE\_EVENT , IR.GetDevice("Device\_Name"), function(type, text)

- **Device\_Name** - the name of the device created in iRidium GUI or iRidiumScript
- **type** - the event type
- **text** - the message from the device

```
IR.AddListener(IR.EVENT_RECEIVE_EVENT , IR.GetDevice("Sonos"), function(type,
text)
// The event is activated when some event is activated on the UPnP device
{
IR.Log(type, text) // Output the event type and text in the log
});
```

## EVENT\_CHANNEL\_SET

The event is activated when the channel is activated.

**Syntax:** IR.AddListener(IR.EVENT\_CHANNEL\_SET , IR.GetDevice("Device\_Name"), function(Name)

- **Device\_Name** - the name of the device created in iRidium GUI or iRidiumScript
- **Name** - the name of the activated channel

```
IR.AddListener(IR.EVENT_CHANNEL_SET, IR.GetDevice("Sonos"), function(Name)
// The event is activated when the channel is activated
{
switch(Name)
{
```

```
case "Volume":
{
    // Commands performed at the activation of the Volume channel

    break;
}
case "Play":
{
    // Commands performed at the activation of the Play channel
    break;
}
}
});
```

[↑ Back](#)

## **EVENT\_APP\_ENTER\_BACKGROUND**

The event is activated when the iRidium application goes to the background mode.

```
IR.AddListener(IR.EVENT_APP_ENTER_BACKGROUND, 0, function()
{
    IR.Log("APP_ENTER_BACKGROUND");
    app_in_background = true;
});
```

[↑ Back](#)

## **EVENT\_APP\_ENTER\_FOREGROUND**

The event is activated when the iRidium application goes to the foreground mode.

```
IR.AddListener(IR.EVENT_APP_ENTER_FOREGROUND, 0, function()
{
    IR.Log("APP_ENTER_FOREGROUND");
    app_in_background = false;
});
```

[↑ Back](#)

## **EVENT\_APP\_WILL\_TERMINATE**

The event is activated before the iRidium application is terminated.

```
IR.AddListener(IR.EVENT_APP_WILL_TERMINATE, 0, function()
{
    IR.Log("APP_WILL_TERMINATE");
```

```
});
```

[↑ Back](#)

## EVENT\_RECIEVE\_NOTIFY

The event is activated when the iRidium application receives a local notification.

**Syntax:** IR.AddListener(IR.EVENT\_RECIEVE\_NOTIFY, 0, function(text, id)

- **text** - the notification text
- **id** - the notification identifier (any text)

```
IR.AddListener(IR.EVENT_RECIEVE_NOTIFY, 0, function(text, id)
{
    if(app_in_background == true)
    {
        IR.Log("SCRIPT_EVENT_RECIEVE_NOTIFY");
        IR.Log(text + " " + id);
        if(id == "D1")
            IR.ShowPopup("Popup 1"); // if the notification with the D1
identifier is received, show Popup 1
        else if(id == "D2")
            IR.ShowPopup("Popup 2"); // if the notification with the D2
identifier is received, show Popup 2
        else if(id == "D3")
            IR.ShowPopup("Popup 3"); // if the notification with the D3
identifier is received, show Popup 3
    }
});
```

[↑ Back](#)