Contents

DOWNLOAD: Example of a project

Description

With iRidiumScript you can:

- Create drivers
- Edit drivers
- Send data to devices
- Receive and process data from devices

Creating Drivers

1. To create drivers with iRidiumScript use the command:

IR.CreateDevice(Device_Type, "Device_Name", "IP", Port);

- **Device_Type** - the protocol type the device works with. At the moment you can use the following types of easily configured protocols:
  - **IR.DEVICE_CUSTOM_TCP**
  - **IR.DEVICE_CUSTOM_UDP**
  - **IR.DEVICE_CUSTOM_HTTP_TCP**
- **Device_Name** - the name of a device (set at random)
- **IP** - the IP-address of the device
- **Port** - the device port which data are received and sent through

2. When the driver is created it is required to connect it to the device with the command:

IR.GetDevice("Device_Name").Connect;

```
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); //The
```

```
command for driver creating
  IR.GetDevice("DEVICE").Connect(); //The command for connecting to the
device
});
```

Editing Drivers

With iRidiumScript you can change any driver properties at any moment. To do that you need to assign the driver to the variable with the command:

IR.GetDevice("Driver_Name");

After that refer to the driver property and set its new value:

driver.Name = "New_Device";

```
IR.AddListener(IR.EVENT_START, 0, function()  //Event is activated at the
application launch
{
  var driver = IR.GetDevice("DEVICE");//Assign the driver to the variable
  driver.Name = "New_Device"; //Change the name of the driver
});
```

Sending Data

With iRidiumScript you can send any required instructions to the device. To do that use the command:

IR.GetDevice("Device_Name").Send([command]);

- **Device_Name** - the name of the device created in iRidium GUI or iRidiumScript
- **command** - the instruction send 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); //The
command for driver creating
  IR.GetDevice("DEVICE").Send(['getverion\r\n HTTP 1.1\r\n\r\n']);//The
command sends instruction to the device('getverion\r\n HTTP 1.1\r\n\r\n')
});
```

Receiving and Processing Data

With iRidiumScript you can receive data from the device with the help of Listener:

IR.AddListener(Information_Display_Type, Device_Name, function(text)

- **Information_Display_Type** - the way the received information is displayed (text format - IR.EVENT_RECEIVE_TEXT or bit format - IR.EVENT_RECEIVE_DATA)
- **Device_Name** - the name of a device created in iRidium GUI or with iRidiumScript

```
DEVICE = IR.GetDevice("Global") //Indicate the name of the device
IR.AddListener(IR.EVENT_RECEIVE_TEXT, DEVICE, function(text) //Event is
activated when receiving data from the device
{
IR.Log(text) //Output the text in the log
});
```

Working with UPnP

UPnP (Universal Plug and Play) is architecture of multirange connections between personal computers and smart devices for example in the home local network. UPnP is built on the basis of Internet standards and technologies such as TCP/IP, HTTP and XML. It provides automatic connection of smart devices to each other and their concurrent work in network environment. These features ensure easy configure of the network (for example, the local network) for more users.

In iRidiumScript you can find, set and control all devices which support UPnP. If you work with UPnP you don't need to know the IP-address and port of your device. The system can find all devices and display them itself.

To create UPnP drivers use the command:

upnpControl = IR.CreateDevice(IR.DEVICE_UPNP_CONTROL, "MyUPnP");

- **MyUPnP** - the name of a UPnP driver
- **IR.DEVICE_UPNP_CONTROL** - Indicate the UPnP type
- **upnpControl** - the variable which the driver identifier is assigned to

Then the driver has to be activated with the command:

upnpControl.Connect();

When launching the Client the driver sends requests to the network about having UPnP devices in it. To manipulate the device found in the network use Listener:

IR.AddListener(IR.EVENT_DEVICE_FOUND, 0, FoundDevice);

- **IR.EVENT_DEVICE_FOUND** - Event is activated in the case of identifying devices which support UPnP
- **FoundDevice** - the function is launched when activating the IR.EVENT_DEVICE_FOUND event

Instructions for Creating Drivers

To create drivers use the object-oriented paradigm. There are several stages for creating drivers:

- **1 Stage** Creating the Main Class

At this stage you are required to create the main class which you will use for creation of driver instances. In iRidiumScript the main class is created as the function:

```
var GlobalCache = function() //Activation of the main class
{
  //The class body
}
```

- **2 Stage** Connection of class instances to the driver in GUI Editor

For each driver you are going to work with you are required to create a base driver in the Device Tree window of GUI Editor for correct operation. After that you are required to describe the connection of each driver instance to the base device (driver) from Device Tree of GUI Editor.

Base drivers are divided by names. So when activating class instances you are required to assign each class instance to its own base driver from Device Tree by its name. In the main class it can be done as follows:

this.DEVICE = IR.GetDevice(this.DriverName);

- **this.DEVICE** - the variable where the driver identifier received by the indicated name is stored
- **this.DriverName** - the variable where the base driver name indicated by the user is stored

```
var GlobalCache = function() //Activation of the main class
{
  this.DriverName; //The driver name set by the user
  this.DEVICE; //Indication to the base driver in Device Tree

  function initalization()//Method for activation of the class instance
  {
    this.DEVICE = IR.GetDevice(this.DriverName); //Defining the indication to
the base driver by its name
  }

  this.Init = initalization; // Defining a method for activation of the
driver instance
}
```

- **3 Stage** Checking the device network status (Online/Offline)

This checking is required so you could see if the device started running or not (if the session of connection to the controlled equipment is started). In the initial state - *that.Online = false* and that means that the device is not in the network but after that the device is launched and the status *that.Online* changes to *that.Online = true*. If the device is not in the network (there is no connection)

you can not send commands to the device.

```
//-------------------------------------------------------
// Handler for device transition to the Online mode
//-------------------------------------------------------
    IR.AddListener(IR.EVENT_ONLINE, that.DEVICE, function(text)
    {
      IR.Log(that.DriverName+" DEVICE is Online");
      that.OnLine = true;
    }, that);
//-------------------------------------------------------
// Handler for device transition to the Offline mode
//-------------------------------------------------------
    IR.AddListener(IR.EVENT_OFFLINE, that.DEVICE, function(text)
    {
      IR.Log(that.DriverName+" DEVICE is Offline");
      that.OnLine = false;
    }, that);
  });
```

- **4 Stage** Sending commands with the help of scripts

With iRidiumScript you can send commands, for example:

```
function SendGetDevice()
{
Device.Send(['getdevices',13]); //Send the "getdevices" command
};

this.sendgetdevice = SendGetDevice

function SendGetVersion()
{
Device.Send(['getversion',13]); //Send the "getversion" command
};

this.sendgetversion = SendGetVersion
```

- **5 Stage** Data output in the log

In iRidiumScript you can output both text and binary data in the log:

```
IR.AddListener(IR.EVENT_RECEIVE_TEXT,0.function(text) //Processing text data
{
IR.Log(text)
});
IR.AddListener(IR.EVENT_RECEIVE_DATA,0.function(text) //Processing binary
data
{
IR.Log(text)
});
```

- **6 Stage** Parser and activation of parser methods (handler of incoming data)

In iRidiumScript you can parse data both through search *(text.indexOf)* and regular expressions *(text.search)*

```
this.IsDeviceRegex       = "DEVICE";
this.IsVersionRegex      = "version";

IR.AddListener(IR.EVENT_RECEIVE_TEXT, this.DEVICE, function(text)
   {

     IR.Log("responce text = "+text);
     that.IsDevice = text.indexOf(that.IsDeviceRegex.toLowerCase());
     that.IsVersion = text.indexOf(that.IsVersionRegex);

      if (that.IsDevice != -1)
     {
       that.getdevices(text);
     } else if (that.IsVersion != -1)
     {
      that.getversion(text);
     }
});
```

- **7 Stage** Referring to functions after the parsing

To refer to functions you are required:

```
this.ResponceGetDevices;
function GetDevices(text)
{
```

```
  var EndPacketRegex = 'endlistdevices';
  var EndPacket = text.indexOf(EndPacketRegex);
  var Responce;
  Responce = text.slice(0,EndPacket);
  that.ResponceGetDevices = Responce;
}
this.getdevices = GetDevices;

this.ResponceGetVersion;
 function GetVersion(text)    //The function for processing the "getversion"
channel
{
  var EndPacket = text.length;
  var Module;
  var Version;
  Module = text.slice(8,9);
  Version = text.slice(10,EndPacket-1);
  that.ResponceGetVersion = "Module: "+Module;
  that.ResponceGetVersion = "Version: "+Version;
}
this.getversion = GetVersion;
```

- **8 Stage** Creating driver instances

```
var [Device_Name] = new GlobalCache();

[Device_Name].DriverName = "Driver_Name(indicated in DEVICE TREE)";

[Device_Name].Init();

var GC06 = new GlobalCache();

GC06.DriverName = "gc-2";

GC06.Init();
```

- **9 Stage** Binding methods and properties to GUI items

```
IR.AddListener(IR.EVENT_ITEM_PRESS,IR.GetItem("Page 1").GetItem("Item
1"),function()
{
  GC06.sendgetversion()
```

```
});
```

DOWNLOAD: Example of a project