# Contents



This section describes the creation process of a freely customizable script driver in **iRidium GUI Editor** through the example of the driver for a **Marantz SR7007** device.

### JS Script driver

it is a driver created on the basis of the native AV & Custom Systems driver with the help of **iRidium DDK**. Scripts enable receiving data and communication with ANY equipment. Driver scripts can be created by the user for operation with any control protocol.

Creation of a **driver** consists of several stages:

# Preparations for Driver Creating

## Selection of Equipment

At the first stage you are required to select a controlled device. It can be target controlled equipment or format converter. As an example the **Marantz SR7007** device is selected. It supports operation by the TCP protocol.

iRidium controlled devices can receive control commands using one of the following protocols:

- **TCP**
- **UDP**
- **RS232**
- **HTTP**

## Search for Documentation

In order to control equipment and receive data from it you are required to find documentation (of API or DDK equipment). Usually you can find the documentation on the web site of the manufacturer but not all manufacturers publish it on open access. If the documentation cannot be found the creation of the driver is not possible.

For the **Marantz SR7007** device the documentation was found on the web site of the manufacturer. You can download it.

## Analysis of the Documentation

When the documentation is received you are required to analyze it and define:

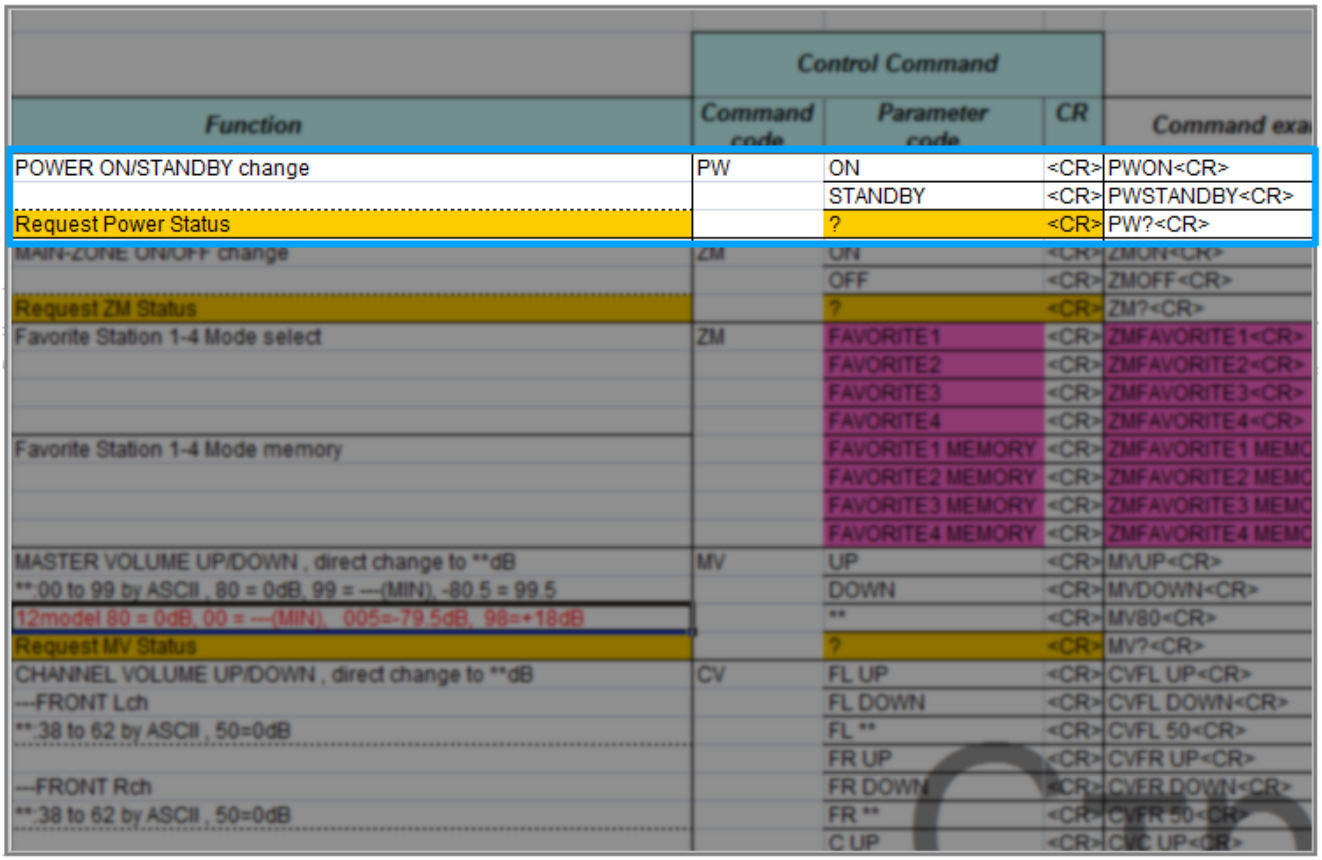
- Control commands for your equipment
- Syntax used by the control commands
- Properties of commands
- Data you can receive from the equipment
- Syntax for receiving data

After analyzing the documentation for **Marantz SR7007** the following conclusions were drawn:

**Syntax:** Commands are formed from two parts (1 part - *command*, 2 part - *property*)

> **Example:** Let us examine the "POWER ON/STANDBY change" command. Its ASCII string looks as **PWON<CR>**.

- The first part **PW** is the command itself (control of the device power);
- The second part **ON** is the property of the command (power feed to the device);
- **<CR>** is a nonprintable character marking the string end (0x0D in the HEX format) .



| Function | Control Command | | | |
|---|---|---|---|---|
| | Command code | Parameter code | CR | Command exa |
| POWER ON/STANDBY change | PW | ON | <CR> | PWON<CR> |
| | | STANDBY | <CR> | PWSTANDBY<CR> |
| Request Power Status | | ? | <CR> | PW?<CR> |
| MAIN-ZONE ON/OFF change | ZM | ON | <CR> | ZMON<CR> |
| | | OFF | <CR> | ZMOFF<CR> |
| Request ZM Status | | ? | <CR> | ZM?<CR> |
| Favorite Station 1-4 Mode select | ZM | FAVORITE1 | <CR> | ZMFAVORITE1<CR> |
| | | FAVORITE2 | <CR> | ZMFAVORITE2<CR> |
| | | FAVORITE3 | <CR> | ZMFAVORITE3<CR> |
| | | FAVORITE4 | <CR> | ZMFAVORITE4<CR> |
| Favorite Station 1-4 Mode memory | | FAVORITE1 MEMORY | <CR> | ZMFAVORITE1 MEMO |
| | | FAVORITE2 MEMORY | <CR> | ZMFAVORITE2 MEMO |
| | | FAVORITE3 MEMORY | <CR> | ZMFAVORITE3 MEMO |
| | | FAVORITE4 MEMORY | <CR> | ZMFAVORITE4 MEMO |
| MASTER VOLUME UP/DOWN , direct change to **dB | MV | UP | <CR> | MVUP<CR> |
| **:00 to 99 by ASCII , 80 = 0dB, 99 = ---(MIN), -80.5 = 99.5 | | DOWN | <CR> | MVDOWN<CR> |
| 12model 80 = 0dB, 00 = ---(MIN), 005=-79.5dB, 98=+18dB | | ** | <CR> | MV80<CR> |
| Request MV Status | | ? | <CR> | MV?<CR> |
| CHANNEL VOLUME UP/DOWN , direct change to **dB | CV | FL UP | <CR> | CVFL UP<CR> |
| ---FRONT Lch | | FL DOWN | <CR> | CVFL DOWN<CR> |
| **:38 to 62 by ASCII , 50=0dB | | FL ** | <CR> | CVFL 50<CR> |
| | | FR UP | <CR> | CVFR UP<CR> |
| ---FRONT Rch | | FR DOWN | <CR> | CVFR DOWN<CR> |
| **:38 to 62 by ASCII , 50=0dB | | FR ** | <CR> | CVFR 50<CR> |
| | | C UP | <CR> | CVC UP<CR> |

This principle is used for forming all commands for the **Marantz SR7007** device.

# Driver Creation

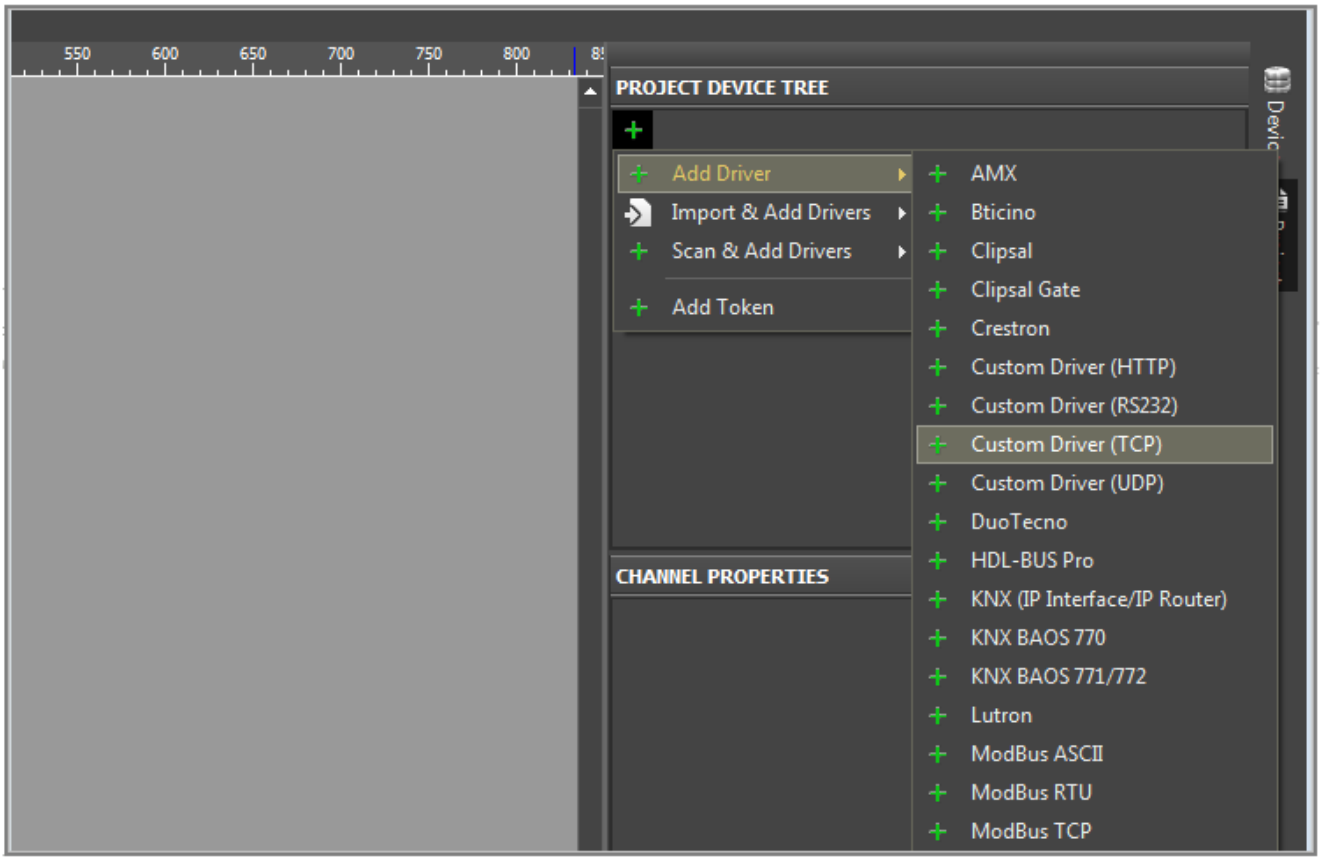
## Driver Creation in GUI Editor

All ready **drivers** are stored in **Device Base** (the global data base). **iRidium GUI Editor** enables you to create a driver both in Device Base – without assignment to a particular project (the driver can be added in any project) and in **Project Device Panel** – the device base of the particular project. Creation of a driver in Project Device Panel enables its launch in the editing mode (emulation and debugging) directly in the project. In order to add such a driver in Device Base you have to do it manually.

In the present example we create the driver in Project Device Panel (with its assignment to the project) to have the possibility of quick editing.

After reading the documentation you can proceed to the driver creation in **Project Device Panel**. **iRidium GUI Editor** has templates of freely customizable drivers which differ by protocols used for data transfer:

- **Custom Driver TCP**
- **Custom Driver UDP**
- **Custom Driver HTTP**
- **Custom Driver RS232**

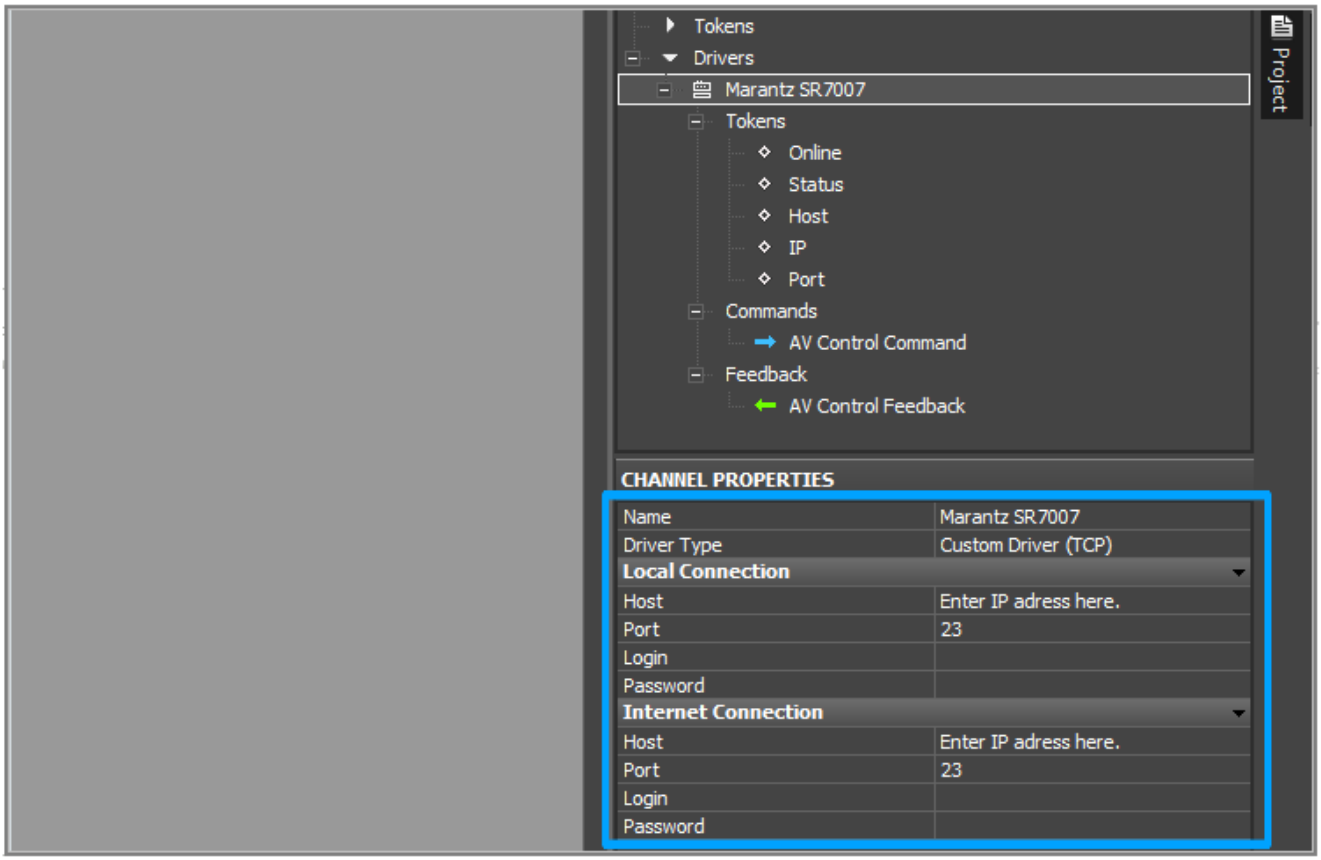



For creation of a new device in the project tree you are required to perform the following actions:

- Create a new project in iRidium GUI Editor
- Go to the **Project Device Panel** tab
- Push the **Add** button or click the right mouse button in the free space of **Project Device Panel**.
- Go to the **Add Driver > Custom Device** section
- The **Marantz SR7007** device can be controlled by the **TCP** protocol so select **Custom Driver TCP**

The driver appeared in **Project Device Panel**. The following actions:

- Select the created driver
- Select the name of the device in the **Name** field in the **Channel Properties**. In our example it is **Marantz SR7007** (it is set at random).
- Indicate **Host** – an IP-address of the device and **Port** – a TCP port the commands to the device are sent through in the **Local Connection** (connection in the local network) section. Indicate port 23 for *Marantz SR7007*.




Each device has drop-down lists:

- **Tokens** - global variables with main properties of the controlled equipment. These properties can be read only (they cannot be changed):
  - *Online* – a state of connection to the controlled system (Online/Offline = 1/0)
  - *Status* – a status of connection to the system (Offline/Connect/Online/Disconnect = 0...3)
  - *Host* – a domain name of the remote system
  - *HostPort* – a port of the remote system iRidium App connects to
  - *IP* – a domain name of the module the App connects to
  - *HostIP* – an IP-address of the remote system the App is connected to
  - *Port* – a local client port the connection to the remote device is established through
- **Commands** - control commands which can be sent to the controlled system
- **Feedback** - variables for receiving and storing data received from the controlled system.

## Driver Structure

For driver creation use the object-oriented paradigm.

The object-oriented paradigm enables you use one and the same driver for several similar devices at the same time by creating *instances*.

On the web site javascript.ru you can find detailed instructions about working with objects in Java Script.

**A driver consists of several parts:**

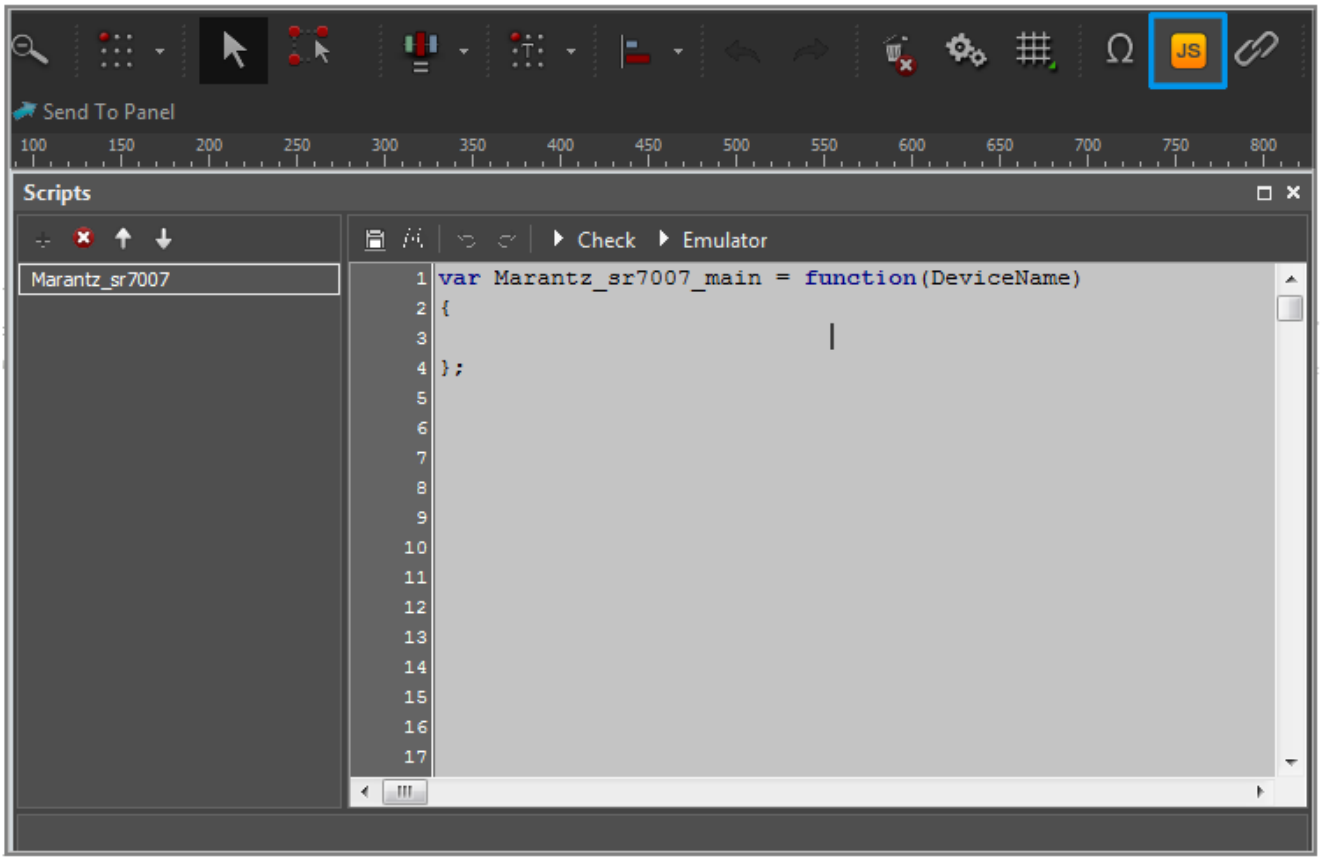
- Main class of the device
  - Section of variables
  - Initalization of the device
  - Event of turning the device on
  - Event of turning the device off
  - Listener of the feedback and data parser
  - Listener of the activated channel
  - Functions of command sending to the device
  - Public functions of the driver
- Creation of an instance

## Writing the Main Class

Open the **Scripts** window in iRidium GUI Editor.

Create a new *script module* with the name of your device. In our example it is **Marantz SR7007**.

The name of the *script module* as the name of the device in Project Device Panel has to be unique. The uniqueness of the name is very important as there can be very many devices and it is recommended to set the name of the module corresponding to the name of the name of the device to avoid the conflict of names.




Let us create the main class you will use later for creating driver instances. In **iRidiumScript** the

main class is created as a function:

```
var Marantz_sr7007_main = function(DeviceName) // Class of drivers for the
Marantz SR7007 device
{
  //body of the class
};
```

**Marantz_sr7007_main** – the name means that the device is **Marantz**, the model is **sr7007**, the class is **main**.

## Section of Variables

Next you are required to indicate the following variables in the class body:

- **this.DriverName** – the name of the device you created in Project Device Panel
- **this.device** – the link to the device created in Project Device Panel
- **this.Online** – the device status. If it is on - **Online = True**, if not - **Online = false**

The DeviceName variable is sent to the function – it will take the device name when creating an instance.

By default It is required to set the 'false' flag to the **this.Online** variable. It means that the device is off.

```
var Marantz_sr7007_main = function(DeviceName) // Class of drivers for the
Marantz SR7007 device
{
  //-------------------------------------------------------
  // Driver Data
  //-------------------------------------------------------
  this.DriverName = DeviceName;
  this.device;
  this.Online = false;
};
```

Further, other variables required for driver work will be added to the section of variables.

## Initalization of the Device

Initalization is the main function of the class. All device listeners, events and functions will be written in it.

Drivers differ by their names so during initalization of class instances it is necessary to assign each instance to its base driver from **Project Device Panel** by its name. So the first thing to be done is to

assign the driver class to the base driver created in **Project Device Panel**.

In order to do this use the **this.device = IR.GetDevice(this.DriverName);** method where the name of the device indicated at the instance creation is set. The result is written in the **this.device** variable.

- **this.device** – the variable storing the driver identifier received by indicated name
- **this.DriverName** – the variable storing the name of the base driver indicated by the user

```
var Marantz_sr7007_main = function() // Class of drivers for the Marantz
SR7007 device
{
  //---------------------------------------------------------
  // Driver Data
  //---------------------------------------------------------
  this.DriverName;
  this.device;
  this.Online = false;

  //---------------------------------------------------------
  // Device Initialization
  //---------------------------------------------------------
  function initialization() // Initalization method of the class instance
  {
    this.device= IR.GetDevice(this.DriverName); // Defining the indicator of
the base driver by its name

    var that = this; //Receiving the link to the object for its using inside
the function
  }
};
```

## Online / Offline Device Events

- **Online** event – the device is on and in the network.
- **Offline** event – the device is off or for any reason there is no connection with it.

These events exist to have the possibility to check if the device is launched or not (if the session of connection to the controlled equipment is opened).

In the initial state **that.Online = false**, which means that the device is offline. But after that the connection starts and the *that.Online* status changes to **that.Online = true**. If the device is offline (the connection is not established), it is impossible to send commands to it.

Add listeners for the Online event in the **initalization()** function:

```
var Marantz_sr7007_main = function() // Class of drivers for the Marantz
SR7007 device
```

```
{
  //---------------------------------------------------------
  // Driver Data
  //---------------------------------------------------------
  this.DriverName;
  this.device;
  this.Online = false;

  //---------------------------------------------------------
  // Device Initialization
  //---------------------------------------------------------
  function initialization() // Initialization method of the class instance
  {
    this.device= IR.GetDevice(this.DriverName); // Defining the indicator of
the base driver by its name

    var that = this; // Receiving the link to the object for its using inside
the function

    //---------------------------------------------------------
    // Device Online
    //---------------------------------------------------------
    IR.AddListener(IR.EVENT_ONLINE, that.DEVICE, function(text)
    {
      IR.Log(that.DriverName+" DEVICE is Online"); // Write to the log that
the device is Online
      that.Online = true; // Assign the true value to the that.Online
variable
    }, that);

    //---------------------------------------------------------
    // Device Offline
    //---------------------------------------------------------
    IR.AddListener(IR.EVENT_OFFLINE, that.DEVICE, function(text)
    {
      IR.Log(that.DriverName+" DEVICE is Offline"); // Write to the log that
the device is Offline
      that.Online = false; // Assign the false value to the that.Online
variable
    }, that);
  }
};
```

## Listener of Feedback

**iRidium Script</b> has the IR.EVENT_RECEIVE_TEXT</b> event which is activated when the device sends data to the application.**

The data can be received by sending the **text** variable for this event to the listener in the function. Then these data can be analyzed and the required information can be extracted - **parser** analyzes

and extracts data.

Let us add a listener for the **IR.EVENT_RECEIVE_TEXT** event in the **initalization()** function:

```
var Marantz_sr7007_main = function() // Class of drivers for the Marantz
SR7007 device
{
  //-----------------------------------------------------
  // Driver Data
  //-----------------------------------------------------
  this.DriverName;
  this.device;
  this.Online = false;

  //-----------------------------------------------------
  // Device Initialization
  //-----------------------------------------------------
  function initialization() // Initialization method of the class instance
  {
    this.device= IR.GetDevice(this.DriverName); // Defining the indicator of
the base driver by its name

    var that = this; // Receiving the link to the object for its using inside
the function
    //-----------------------------------------------------
    // Device Online
    //-----------------------------------------------------
    IR.AddListener(IR.EVENT_ONLINE, that.DEVICE, function(text)
    {
      IR.Log(that.DriverName+" DEVICE is Online"); // Write to the log that
the device is
Online
      that.Online = true; // Assign the true value to the that.Online
variable
    }, that);

    //-----------------------------------------------------
    // Device Offline
    //-----------------------------------------------------
    IR.AddListener(IR.EVENT_OFFLINE, that.DEVICE, function(text)
    {
      IR.Log(that.DriverName+" DEVICE is Offline"); // Write to the log that
the device is Offline
      that.Online = false; // Assign the false value to the that.Online
variable
    }, that);

    //-----------------------------------------------------
    // Receive Text
```

```
//-------------------------------------------------------
IR.AddListener(IR.EVENT_RECEIVE_TEXT,that.device, function(text)
{
  IR.Log("receive = "+ text);  // Output the received data in the log
});
```

## Parser (Handler of Incoming Data)

The task of ‘‘‘parser’’’ is to extract the required information from the received data and store it for its further output in the application.

**Java Script** provides several functions for data extracting and search in string variables:

- .indexOf()
- .lastIndexOf()
- .search()
- .slice()

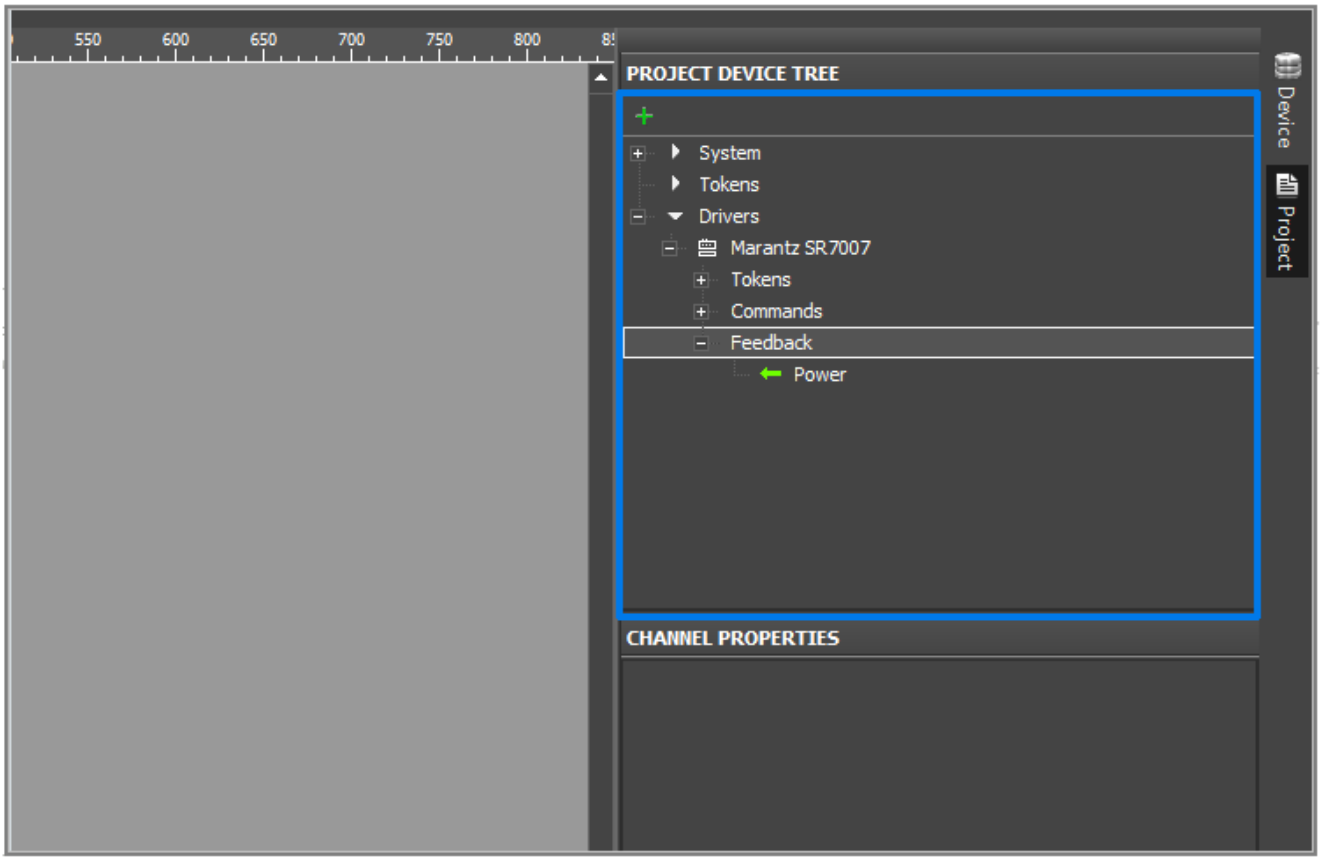
**Parser** is written in the listener for the **IR.EVENT_RECEIVE_TEXT** event. In special cases it is written in a separate function but it still can be launched only inside the listener.

**Marantz SR7007** can send the string consisting of the command name and its state. The first two symbols are the command, for example **PW**. The rest part is the state, for example **ON** or **OFF**.

Searching for all command variants of is performed by the ‘‘‘Switch’’’ function. When you found the command it is required to store its state in the corresponding variable inside the object and in the variable of the Feedback section(the variables are created manually in the project tree and their names have to be unique).

- Tell the **this.PowerStatus** variable in the section of variables
- Store the received state in the variable in the **case "PW"** section inside the parser.

- Create the **Power** variable in **Project Device Panel**, the **Feedback** section.
- Copy the received state in the **Power** variable with the help of the **IR.SetVariable("Drivers."+that.DriverName+".Power", answer);** command.



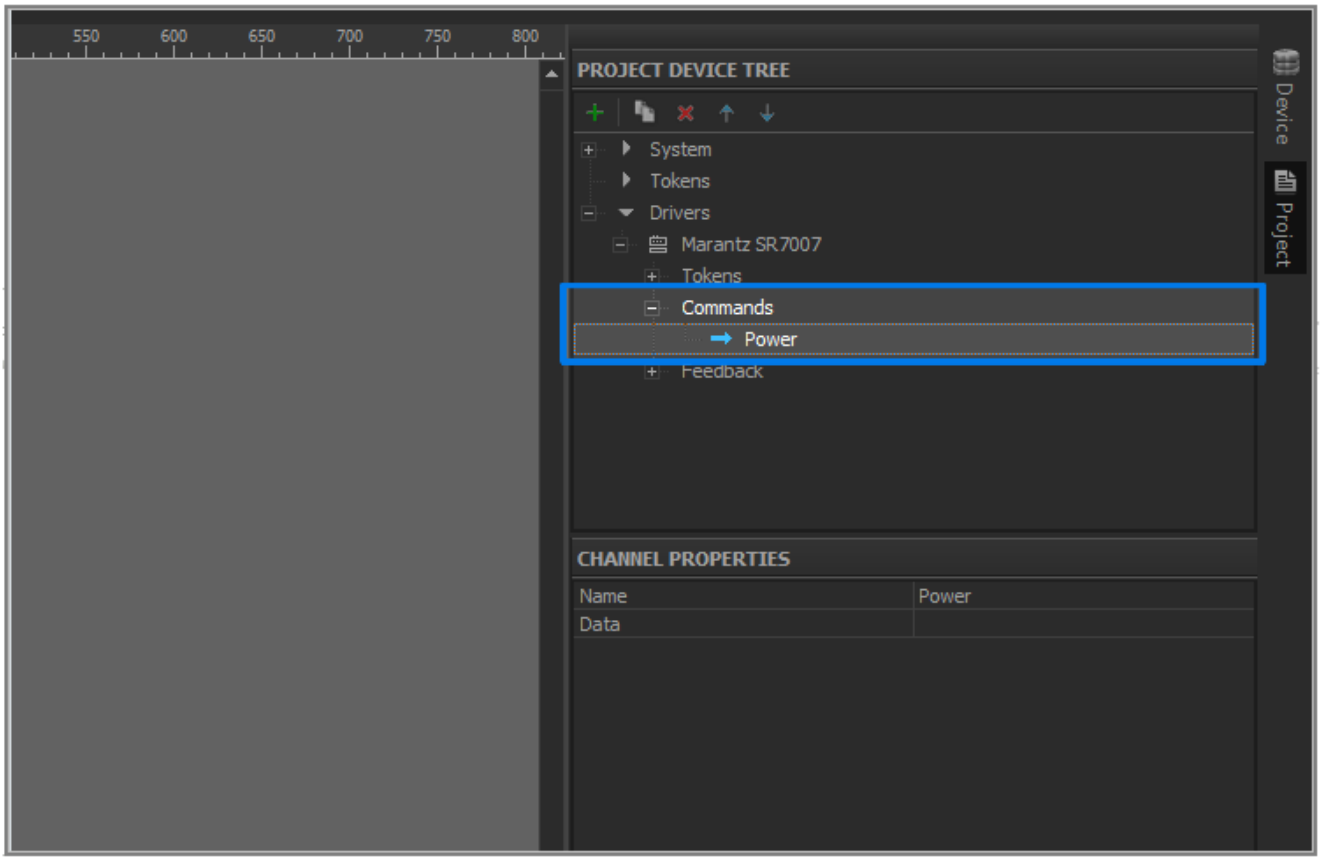

```
var Marantz_sr7007_main = function() // Class of drivers for the Marantz
SR7007 device
{
  //------------------------------------------------------
  // Driver Data
  //------------------------------------------------------
  this.DriverName;
  this.device;
  this.Online = false;

  this.PowerStatus; // It will store the state of the Power command

  //------------------------------------------------------
  // Device Initialization
  //------------------------------------------------------
  function initialization() // Initialization method of the class instance
  {
    this.device= IR.GetDevice(this.DriverName); // Defining the indicator of
the base driver by its name

    var that = this; // Receiving the link to the object for its using inside
the function
    //------------------------------------------------------
    // Device Online
    //------------------------------------------------------
    IR.AddListener(IR.EVENT_ONLINE, that.DEVICE, function(text)
```

```
    {
      IR.Log(that.DriverName+" DEVICE is Online"); // Write to the log that
the device is Online
      that.Online = true; // Assign the true value to the that.Online
variable
    }, that);

    //---------------------------------------------------------
    // Device Offline
    //---------------------------------------------------------
    IR.AddListener(IR.EVENT_OFFLINE, that.DEVICE, function(text)
    {
      IR.Log(that.DriverName+" DEVICE is Offline"); // Write to the log that
the device is Offline
      that.Online = false; // Assign the false value to the that.Online
variable
    }, that);

    //---------------------------------------------------------
    // Receive Text
    //---------------------------------------------------------
    IR.AddListener(IR.EVENT_RECEIVE_TEXT,that.device, function(text)
    {
      IR.Log("receive = "+ text);  // Output the received data in the log

      // Parser
      var cmd = text.slice(0,2); // Cut out the first two symbols – the
command name – from the received data
      var answer = text.slice(2,text.length-1); // Cut out the rest – the
current state
      IR.Log("cmd = "+cmd); // Output the cutout command
      IR.Log("answer = "+answer); // Output the current state
      switch (cmd)
      {
        case "PW":
           that.PowerStatus = answer;
           /* Write the current state in the that.PowerStatus variable for
            *   storing the status inside the object */
           IR.SetVariable("Drivers."+that.DriverName+".Power", answer);
           // Write the current state in Feedback for outputting the state in
the application
           break;
      // This principle can be used for parsing all data received from
Marantz SR7007
      }
    });
   }
};
```

## Listener of the Activated Channel

All *commands* – data for sending to the equipment have to differ by their names.

Add the **Power** command to **Project Device Panel**, the **Commands** section.




When at pressing on an item or otherwise some command is activated in the **iRidium** application, the **IR.EVENT_CHANNEL_SET** event is activated. After assigning **the listener** for this event the creator of the driver is enabled (depending on what command is activated) to form and send the corresponding command to the device. The name of the activated *command* is stored in the **name** variable which is sent together with the function to the listener of the **IR.EVENT_CHANNEL_SET** event.

**IR.AddListener(IR.EVENT_CHANNEL_SET,that.device,function(name)**

When analyzing the documentation for **Marantz SR7007** the conclusion was drawn that the command is formed from two parts and nonprintable symbol.

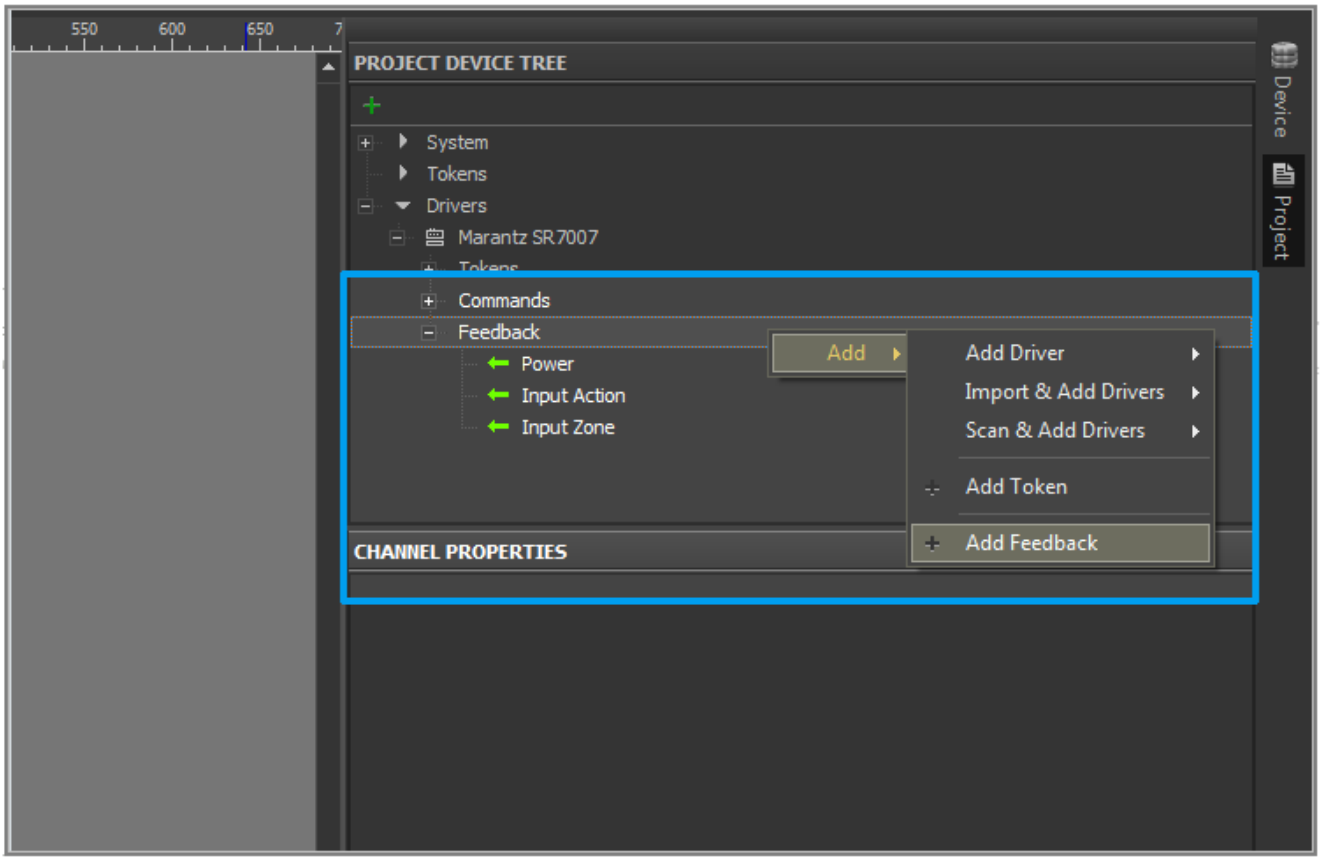
In order to make **the driver** more **flexible** and easily upgradable we will form the command from parts.
The basis for the command is the activated **channel**. The channel defines which *properties* are possible for the command in this particular case. When activating the command the user is required to indicate *properties* for the command to send the equipment data corresponding to the command.

For example, the user has **Marantz SR7007** in **Zone 1** and he/she needs to turn it on. In order to do this it is required to:

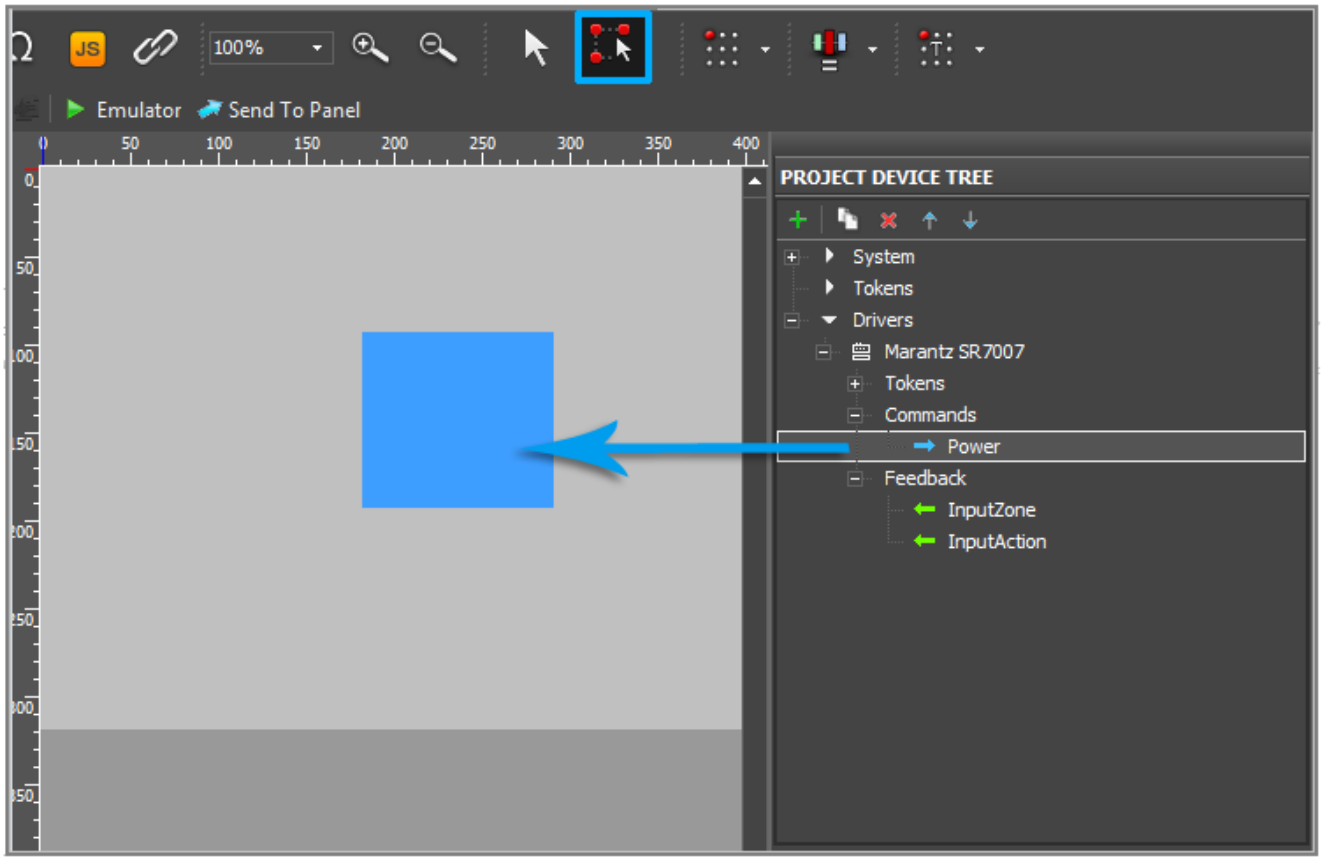
- Activate the channel - **Power**
- Indicate the first property - the zone name - **Zone 1**
- Indicate the second property – the command property - **ON**

Properties for the command will be input in the variables of the **Feedback** section. So add the following variables in the **Feedback** section of **Project Device Panel**:

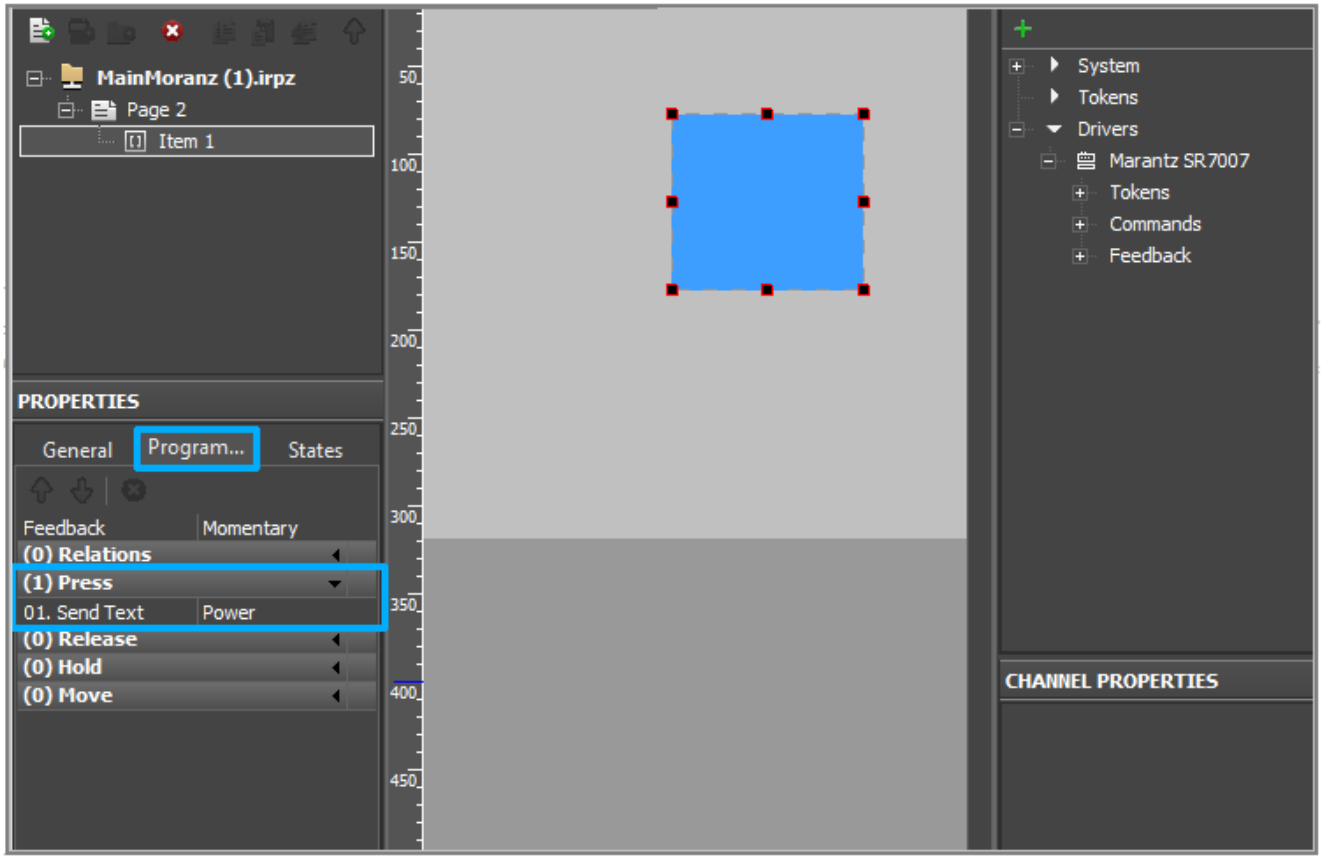
- **Input Action** – it will be used to input a command property, for example **ON**
- **Input Zone** – it will be used to input the zone name, for example - **Zone 1**




- As an example of sending properties, create an item in **GUI Editor**.
- Drag the existing channel **Power** to the item. When pressing on the item the **IR.EVENT_CHANNEL_SET** event is activated and the name of **the channel** which activated the present event will be known in the **script**.

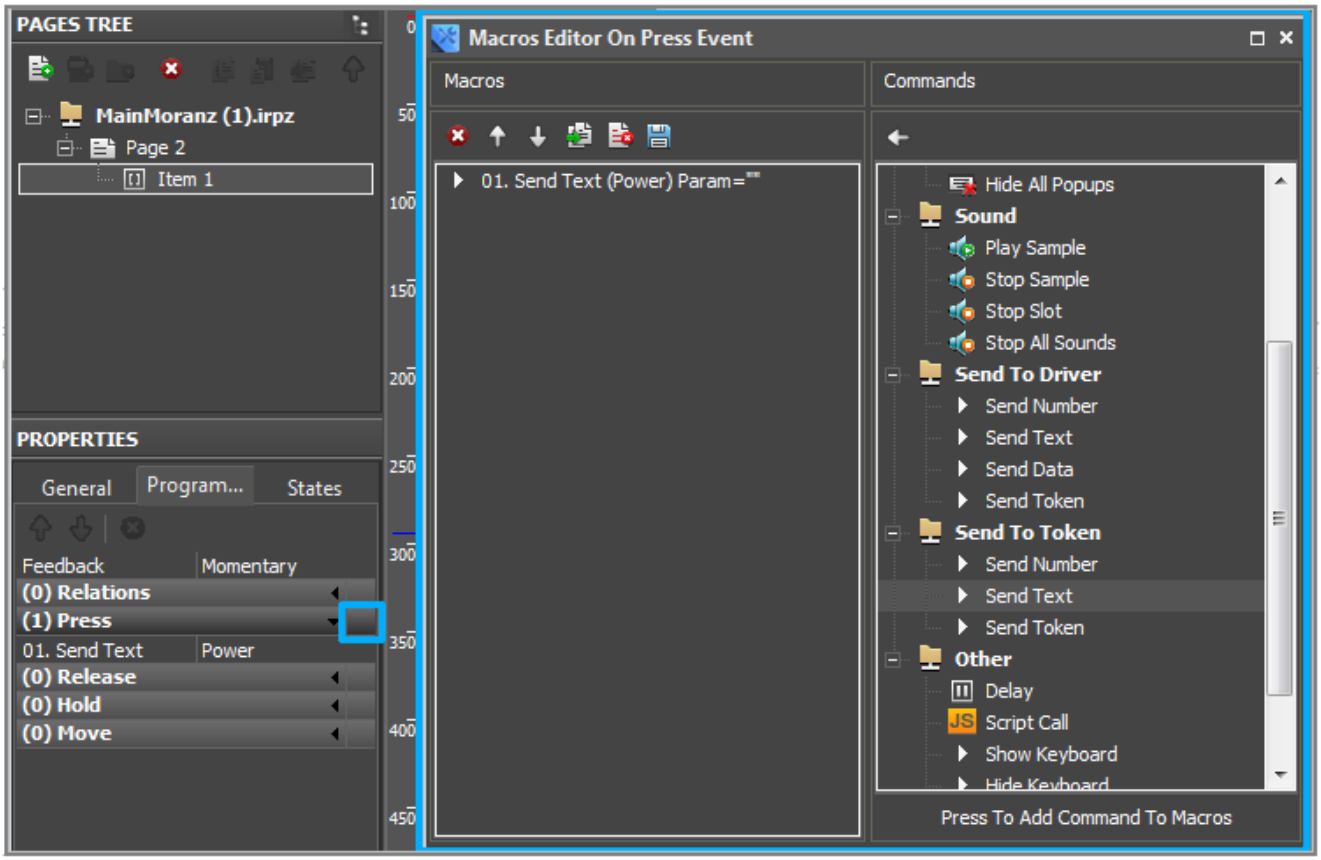



- Select the created item and go to the **Object Properties** panel

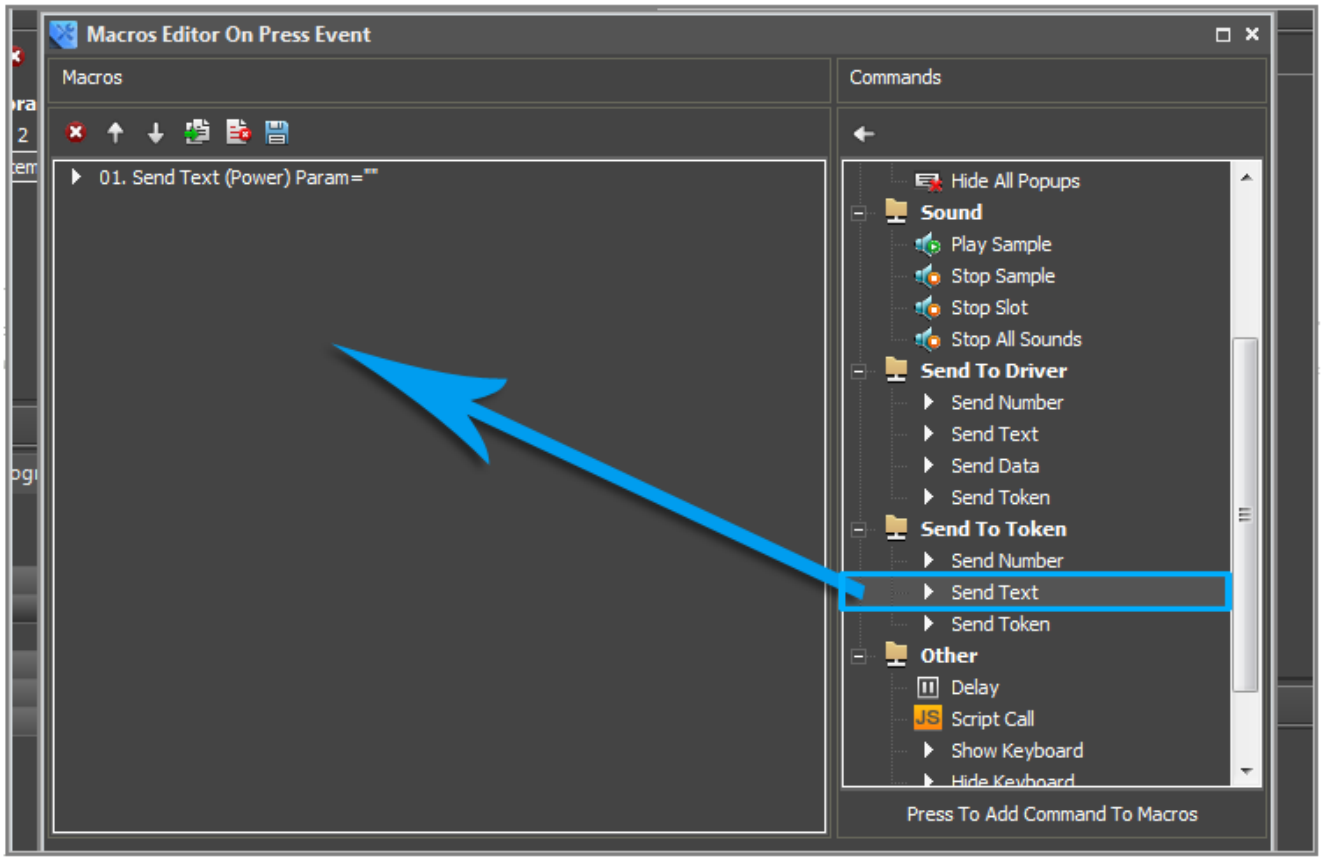



- Open the **Programming** tab. Here you can see **the channel** dragged by you to the item. It is

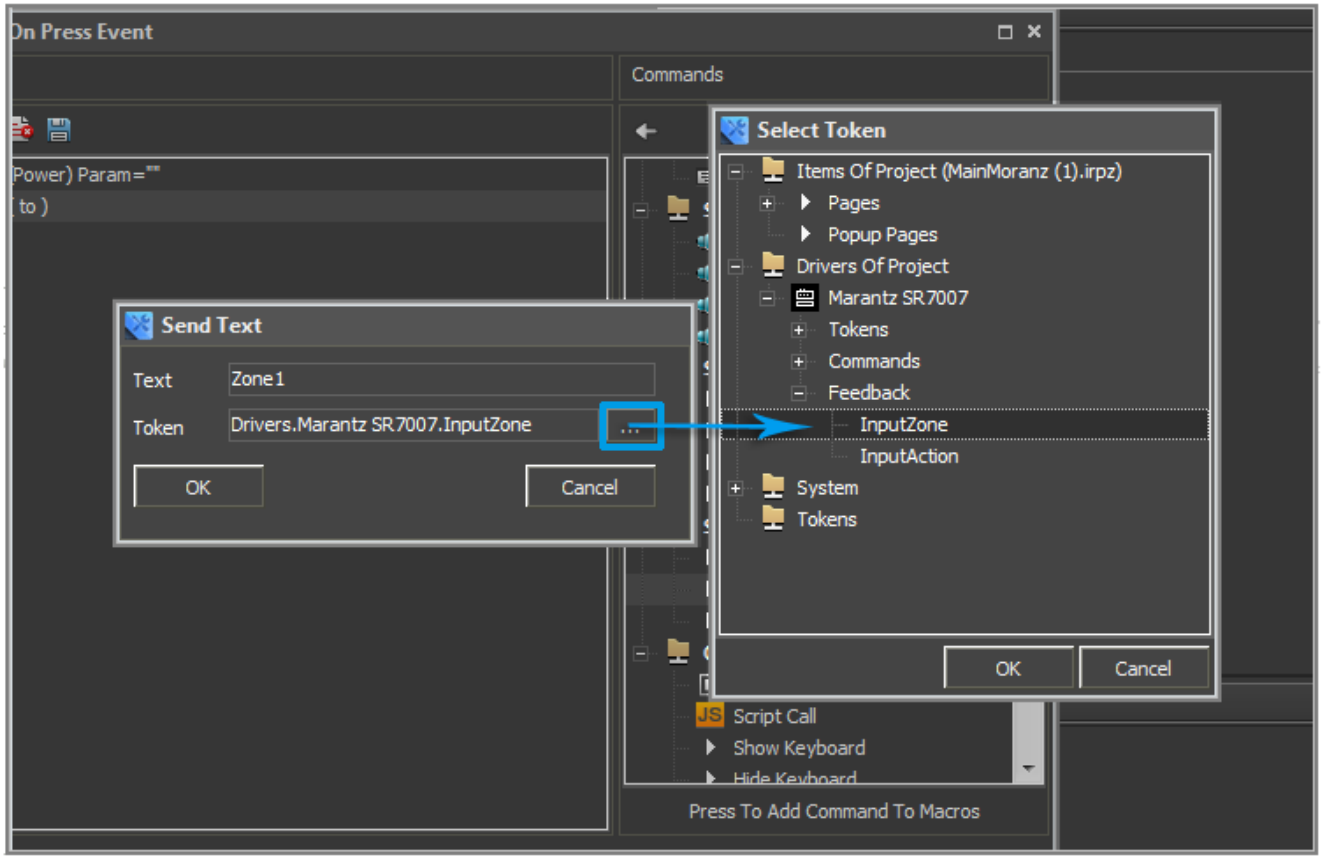
added to the **Press** event. Open the **Macros** windows for the **Press** event.



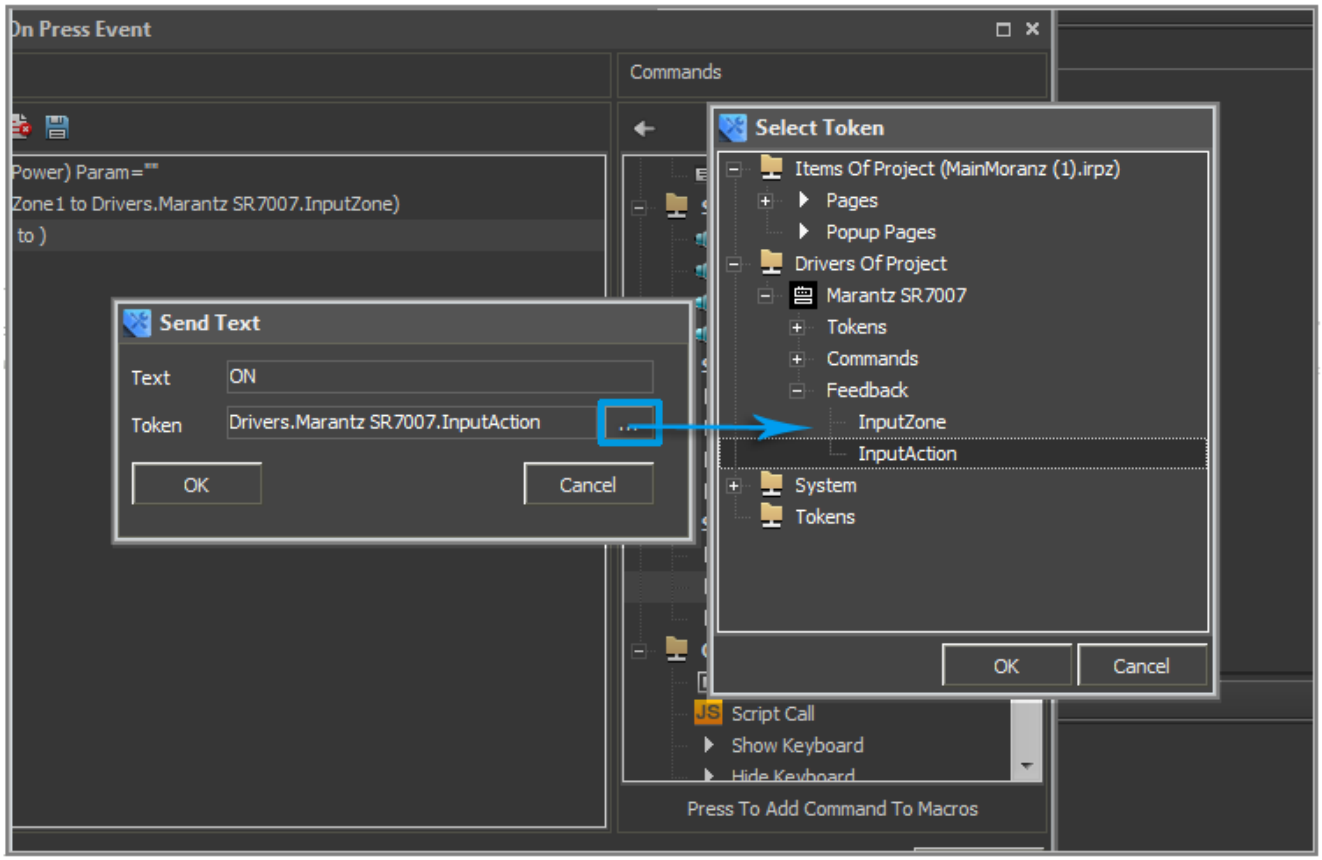

- In the **Commands** column from the **Send To Token** section select **Send Text** and drag in the **Macros** column.



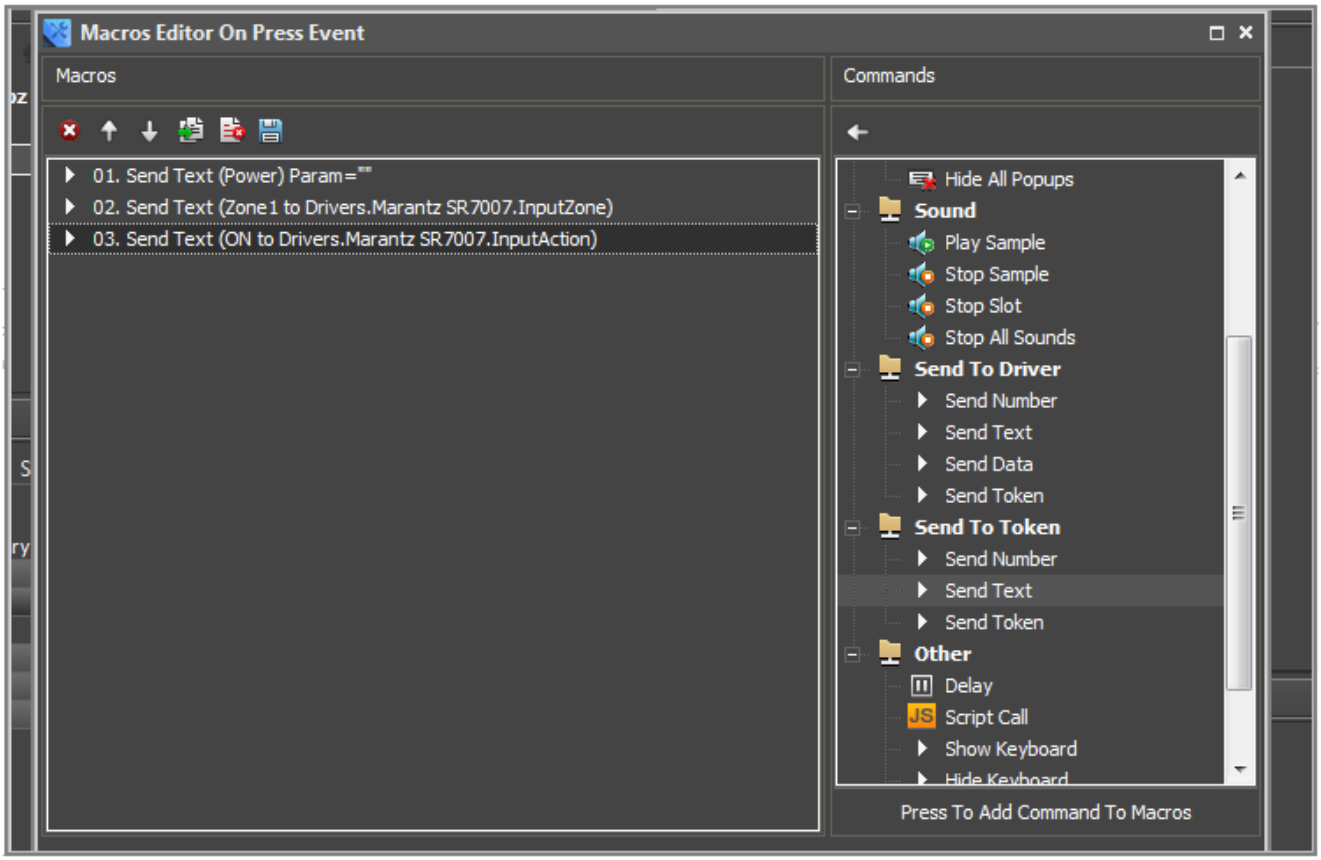

- In the appeared window in the **Text** field input the first property for the command - **the zone name**, in our case it is **Zone1**. In the **Token** field select the variable created in the **Feedback** section - **InputZone**.



- Drag one more macros **Send Text** and in the **Text** field input the second property - **ON**. In the **Token** field select the variables created in the **Feedback** section - **Input Action**.




With the help of the **Send Text** macro you can send any properties to variables in the **Feedback** section.




To receive values from variables in the **Feedback** section there is a command **IR.GetVariable("Drivers."+that.DriverName+".**Variable name**");**

To store and use the received data in the driver input in the **Feedback** section, add the following variables in the section of variables:

- **this.param1** – this variable will store data input by the user in Feedback - **Input Action**
- **this.param2** – this variable will store data input by the user in Feedback - **Input Zone**
- **this.Msg = ""** – this variable will store the string to be sent to the device
- **this.error = false** – the flag variable. If the **param1** ore **param2** properties are set incorrectly, then **error** will be **true** and nothing will be sent to the device. If all properties are set correctly - **error** will be **false** and the command from the **this.Msg** variable will be sent to the device:

```
var Marantz_sr7007_main = function() // Class of drivers for the Marantz
SR7007 device
{
  //-------------------------------------------------------
  // Driver Data
  //-------------------------------------------------------
  this.DriverName;
  this.device;
  this.Online = false;

  this.param1; // Parameter storing the zone name
  this.param2; // Command parameter
  this.Msg = ""; // Variable storing the command for sending to the device
```

```
  this.error = false; // Error flag

  this.PowerStatus; // It will store the state of the Power command

  //---------------------------------------------------------
  // Device Initialization
  //---------------------------------------------------------
  function initialization() // Initialization method of the class instance
  {
    this.device= IR.GetDevice(this.DriverName); // Defining the indicator of
the base driver by its name


    var that = this; // Receiving the link to the object for its using inside
the function

    //---------------------------------------------------------
    // Device Online
    //---------------------------------------------------------
    IR.AddListener(IR.EVENT_ONLINE, that.DEVICE, function(text)
    {
      IR.Log(that.DriverName+" DEVICE is Online"); // Write to the log that
the device is Online
      that.Online = true; // Assign the true value to the that.Online
variable
    }, that);

    //---------------------------------------------------------
    // Device Offline
    //---------------------------------------------------------
    IR.AddListener(IR.EVENT_OFFLINE, that.DEVICE, function(text)
    {
      IR.Log(that.DriverName+" DEVICE is Offline"); // Write to the log that
the device is Offline
      that.Online = false; // Assign the false value to the that.Online
variable
    }, that);

    //---------------------------------------------------------
    // Receive Text
    //---------------------------------------------------------
    IR.AddListener(IR.EVENT_RECEIVE_TEXT,that.device, function(text)
    {
      IR.Log("receive = "+ text);  // Output the received data in the log

      // Parser
      var cmd = text.slice(0,2); // Cut out the first two symbols – the
command name – from the received data
      var answer = text.slice(2,text.length-1); // Cut out the rest – the
current state      IR.Log("cmd = "+cmd); // Output the cutout command
      IR.Log("answer = "+answer); // Output the current state
```

```
      switch (cmd)
      {
        case "PW":
           that.PowerStatus = answer;
           /* Write the current state in the that.PowerStatus variable for
            *   storing the status inside the object */
           IR.SetVariable("Drivers."+that.DriverName+".Power", answer);
           // Write the current state in Feedback for outputting the state in
the application
           break;
      // This principle can be used for parsing all data received from
Marantz SR7007
      }
    });
    //---------------------------------------------------------
    // Channel Set
    //---------------------------------------------------------
    IR.AddListener(IR.EVENT_CHANNEL_SET,that.device,function(name)
    {
       that.Msg = ""; // Clear the command for sending
       that.error = false; // Set the error flag to false
       that.param1 = IR.GetVariable("Drivers."+that.DriverName+".InputZone");
       // Get the zone name
       that.param2 =
IR.GetVariable("Drivers."+that.DriverName+".InputAction");
       // Get the command property
       switch(name) // Search for the channel name
       {
         case "Power":
          Power(name,that.param1,that.param2)
          // The Power  function will form the command for the device and
send it
          // The name of the activated channel , zone name and command
property are sent to it
         break;
       };
    });
   }
};
```

## Functions of Sending Commands to Devices

As an example of a function off sending commands to the device we will use the **Power (name,that.param1,that.param2)** function. It has to:

- form a command
- send the command
- if the input parameters are wrong – output the error message in the log

The **Power** functions receives three variables which will be the basis for command forming:

- **name** – a channel name

- **that.param1** – a zone name
- **that.param2** – a command property

For the **Power** command the **Marantz SR7007** device supports three zones - **Zone1**, **Zone2**, **Zone3** and the common zone - **System**.

Depending on the zone number indicated in the *Input Zone* variable the function has to form a command in the that.Msg variable with parameter **ZM** for Zone1, **Z2** for **Zone2** and **PW** for **System**. If something else is indicated the function shows that there is no such zone.

The same with the command parameter: if the user indicated the **On** parameter – the function should add to the **that.Msg** variable - **ON**, if the user indicated **Off** – it should add **OFF**, in other cases it should output the error in the log.

After adding the zone name and the command parameter the command for **Marantz SR7007** should end with **<CR>**, that is why we will add **<CR>** (the string end) in the **that.Msg** variable.

```
var Marantz_sr7007_main = function() // Class of drivers for the Marantz
SR7007 device
{
  //---------------------------------------------------------
  // Driver Data
  //---------------------------------------------------------
  this.DriverName;
  this.device;
  this.Online = false;

  this.param1; // Parameter storing the zone name
  this.param2; // Command parameter
  this.Msg = ""; // Variable storing the command for sending to the device
  this.error = false; // Error flag

  this.PowerStatus; // It will store the state of the Power command

  //---------------------------------------------------------
  // Device Initialization
  //---------------------------------------------------------
  function initialization() // Initialization method of the class instance
  {
    this.device= IR.GetDevice(this.DriverName); // Defining the indicator of
the base driver by its name

    var that = this; // Receiving the link to the object for its using inside
the function
    //---------------------------------------------------------
    // Device Online
    //---------------------------------------------------------
    IR.AddListener(IR.EVENT_ONLINE, that.DEVICE, function(text)
    {
      IR.Log(that.DriverName+" DEVICE is Online"); // Write to the log that
```

```
the device is Online
      that.Online = true; // Assign the true value to the that.Online
variable
    }, that);

    //---------------------------------------------------------
    // Device Offline
    //---------------------------------------------------------
    IR.AddListener(IR.EVENT_OFFLINE, that.DEVICE, function(text)
    {
      IR.Log(that.DriverName+" DEVICE is Offline"); // Write to the log that
the device is Offline
      that.Online = false; // Assign the false value to the that.Online
variable
    }, that);

    //---------------------------------------------------------
    // Receive Text
    //---------------------------------------------------------
    IR.AddListener(IR.EVENT_RECEIVE_TEXT,that.device, function(text)
    {
      IR.Log("receive = "+ text);  // Output the received data in the log

      // Parser
      var cmd = text.slice(0,2); // Cut out the first two symbols – the
command name – from the received data
      var answer = text.slice(2,text.length-1); // Cut out the rest – the
current state      IR.Log("cmd = "+cmd); // Output the cutout command
      IR.Log("answer = "+answer); // Output the current state
      switch (cmd)
      {
        case "PW":
           that.PowerStatus = answer;
           /* Write the current state in the that.PowerStatus variable for
            *   storing the status inside the object */
           IR.SetVariable("Drivers."+that.DriverName+".Power", answer);
           // Write the current state in Feedback for outputting the state in
the application
           break;
      // This principle can be used for parsing all data received from
Marantz SR7007
      }
    });
    //---------------------------------------------------------
    // Channel Set
    //---------------------------------------------------------
    IR.AddListener(IR.EVENT_CHANNEL_SET,that.device,function(name)
    {
       that.Msg = ""; // Clear the command for sending
       that.error = false; // Set the error flag to false
       that.param1 = IR.GetVariable("Drivers."+that.DriverName+".InputZone");
```

```
       // Get the zone name
       that.param2 =
IR.GetVariable("Drivers."+that.DriverName+".InputAction");
       // Get the command property
       switch(name) // Search for the channel name
       {
         case "Power":
          Power(name,that.param1,that.param2)
          // The Power  function will form the command for the device and
send it
          // The name of the activated channel , zone name and command
property are sent to it
         break;
    };
    });

    //---------------------------------------------------------
    // Command Power
    //---------------------------------------------------------
    function Power(type,zone,action) // The Power function
    {
       switch(that.param1) // Search for the zone
          {
            case "Zone1":
              that.Msg+="ZM"; //Adding the zone name to the command for
sending
            break;
            case "Zone2":
              that.Msg+="Z2";
            break;
            case "Zone3":
              that.Msg+="Z3";
            break;
            case "System":
              that.Msg+="PW";
            break;
            default: //If the zone is not found – to output the error
               IR.Log("error in parameter 1: There are no zone with that
number");
               that.error = true; // Set the true flag for error – it will
stop the command sending
            break;
          };
          that.Msg+=""
          switch(that.param2) // Search for the parameter
          {
            case "On":
              that.Msg+="ON" // Adding the parameter
            break;
            case "Off":
              that.Msg+="OFF"
```

```
            break;
            default: // If the parameter is not correct –output the error
   IR.Log("error in parameter 2: you can't use "+that.param2+" with "+type);
               that.error = true;
            break;
          };
          that.Msg+="<CR>"; // Add the symbol of the string end
          if (that.error == false) // If there are no errors – send the
command to the device
              that.device.Send([that.msg]);
    };
   }
};
```

## Public Functions of the Driver

*Public* functions when working with iRidium Script are the functions available to the user. We added the **Power** function inside the driver class: it sends commands to the device but it works correctly only when *the channel* is activated. The user who will use the driver should not use this function outside the class. So this function is not *public*.

The **initialization** function is the main function of *the class* and it is necessary to perfom it after creating the instance to activate the driver functions. Let us make it public. Below the **initialization** function add **this.Init = initialization**. After creating of an instance it will enable you to activate the **initialization** function with the help of **.Init**:

```
var Marantz_sr7007_main = function() // Class of drivers for the Marantz
SR7007 device
{
  //-------------------------------------------------------
  // Driver Data
  //-------------------------------------------------------
  this.DriverName;
  this.device;
  this.Online = false;

  this.param1; // Parameter storing the zone name
  this.param2; // Command parameter
  this.Msg = ""; // Variable storing the command for sending to the device
  this.error = false; // Error flag

  this.PowerStatus; // It will store the state of the Power command

  //-------------------------------------------------------
  // Device Initialization
  //-------------------------------------------------------
  function initialization() // Initialization method of the class instance
  {
    this.device= IR.GetDevice(this.DriverName); // Defining the indicator of
```

```
the base driver by its name

    var that = this; // Receiving the link to the object for its using inside
the function
    //--------------------------------------------------------
    // Device Online
    //--------------------------------------------------------
    IR.AddListener(IR.EVENT_ONLINE, that.DEVICE, function(text)
    {
      IR.Log(that.DriverName+" DEVICE is Online"); // Write to the log that
the device is Online
      that.Online = true; // Assign the true value to the that.Online
variable
    }, that);

    //--------------------------------------------------------
    // Device Offline
    //--------------------------------------------------------
    IR.AddListener(IR.EVENT_OFFLINE, that.DEVICE, function(text)
    {
      IR.Log(that.DriverName+" DEVICE is Offline"); // Write to the log that
the device is Offline
      that.Online = false; // Assign the false value to the that.Online
variable
    }, that);

    //--------------------------------------------------------
    // Receive Text
    //--------------------------------------------------------
    IR.AddListener(IR.EVENT_RECEIVE_TEXT,that.device, function(text)
    {
      IR.Log("receive = "+ text);  // Output the received data in the log

      // Parser
      var cmd = text.slice(0,2); // Cut out the first two symbols – the
command name – from the received data
      var answer = text.slice(2,text.length-1); // Cut out the rest – the
current state      IR.Log("cmd = "+cmd); // Output the cutout command
      IR.Log("answer = "+answer); // Output the current state
      switch (cmd)
      {
        case "PW":
           that.PowerStatus = answer;
           /* Write the current state in the that.PowerStatus variable for
            *   storing the status inside the object */
           IR.SetVariable("Drivers."+that.DriverName+".Power", answer);
           // Write the current state in Feedback for outputting the state in
the application
           break;
      // This principle can be used for parsing all data received from
Marantz SR7007
```

```
      }
    });
    //---------------------------------------------------------
    // Channel Set
    //---------------------------------------------------------
    IR.AddListener(IR.EVENT_CHANNEL_SET,that.device,function(name)
    {
       that.Msg = ""; // Clear the command for sending
       that.error = false; // Set the error flag to false
       that.param1 = IR.GetVariable("Drivers."+that.DriverName+".InputZone");
       // Get the zone name
       that.param2 =
IR.GetVariable("Drivers."+that.DriverName+".InputAction");
       // Get the command property
       switch(name) // Search for the channel name
       {
         case "Power":
          Power(name,that.param1,that.param2)
          // The Power  function will form the command for the device and
send it
          // The name of the activated channel , zone name and command
property are sent to it
         break;
    };
    });

    //---------------------------------------------------------
    // Command Power
    //---------------------------------------------------------
    function Power(type,zone,action) // The Power function
    {
       switch(that.param1) // Search for the zone
          {
            case "Zone1":
              that.Msg+="ZM"; //Adding the zone name to the command for
sending
            break;
            case "Zone2":
              that.Msg+="Z2";
            break;
            case "Zone3":
              that.Msg+="Z3";
            break;
            case "System":
              that.Msg+="PW";
            break;
            default: //If the zone is not found – to output the error
               IR.Log("error in parameter 1: There are no zone with that
number");
               that.error = true; // Set the true flag for error – it will
stop the command sending
```

```
            break;
          };
          that.Msg+=""
          switch(that.param2) // Search for the parameter
          {
            case "On":
              that.Msg+="ON" // Adding the parameter
            break;
            case "Off":
              that.Msg+="OFF"
            break;
            default: // If the parameter is not correct –output the error
   IR.Log("error in parameter 2: you can't use "+that.param2+" with "+type);
               that.error = true;
            break;
          };
          that.Msg+="<CR>"; // Add the symbol of the string end
          if (that.error == false) // If there are no errors – send the
command to the device
              that.device.Send([that.msg]);

    };
   }

   //--------------------------------------------------------
   // Public
   //--------------------------------------------------------
   this.Init = initialization; // Make the initialization function public
};
```

## Creation of Driver Instances

After the driver class is written you can create an instance. The instance is created below the written class or in another module:

```
var myMarantz = new Marantz_sr7007_main("Marantz SR7007"); // Creation of the
driver instance
```

After creating of the instance activate the **public** function of **initialization**.

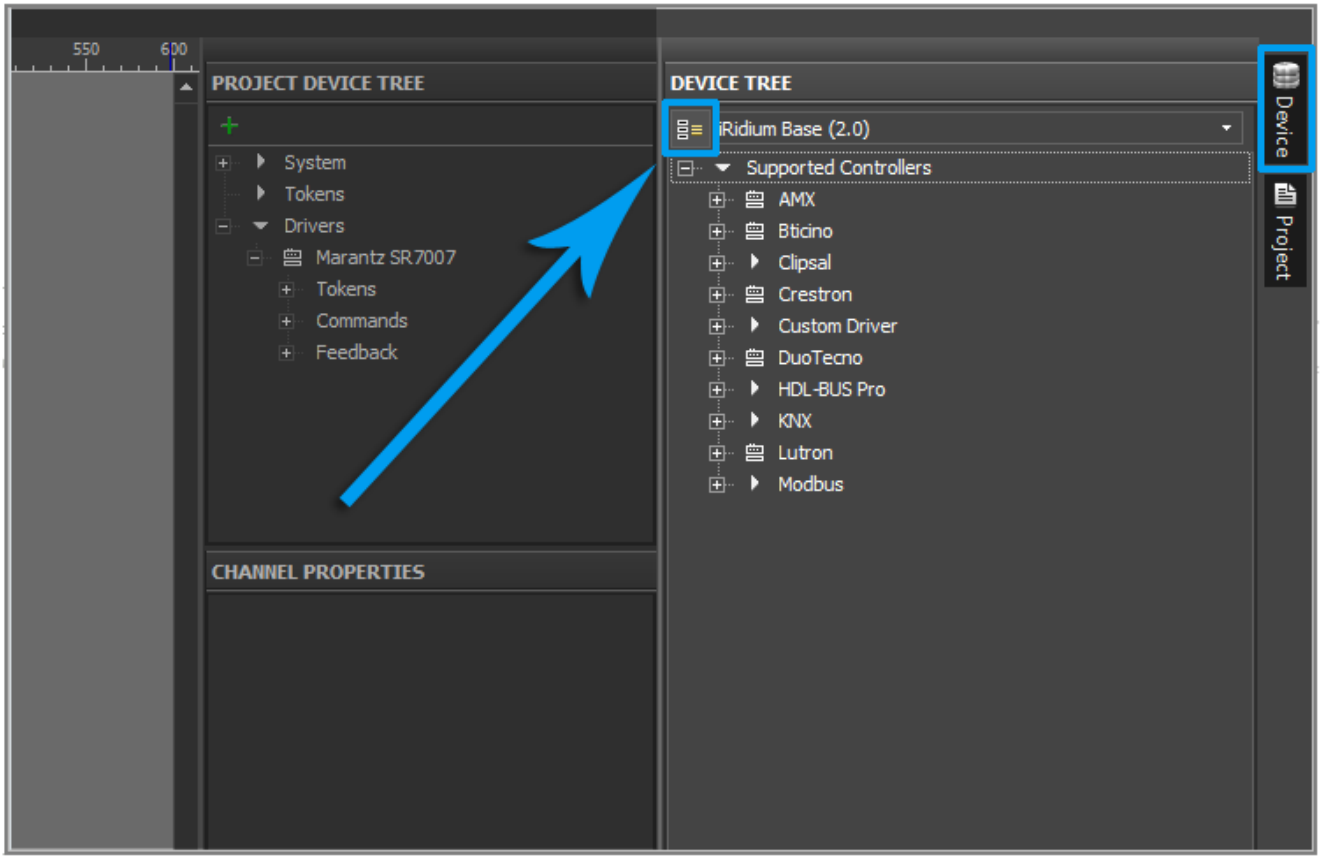
```
var myMarantz = new Marantz_sr7007_main("Marantz SR7007"); // Creation of the
driver instance
myMarantz.Init();
```

After initialization you can launch the project for debugging and testing the driver.

## Adding the Driver in Device Base

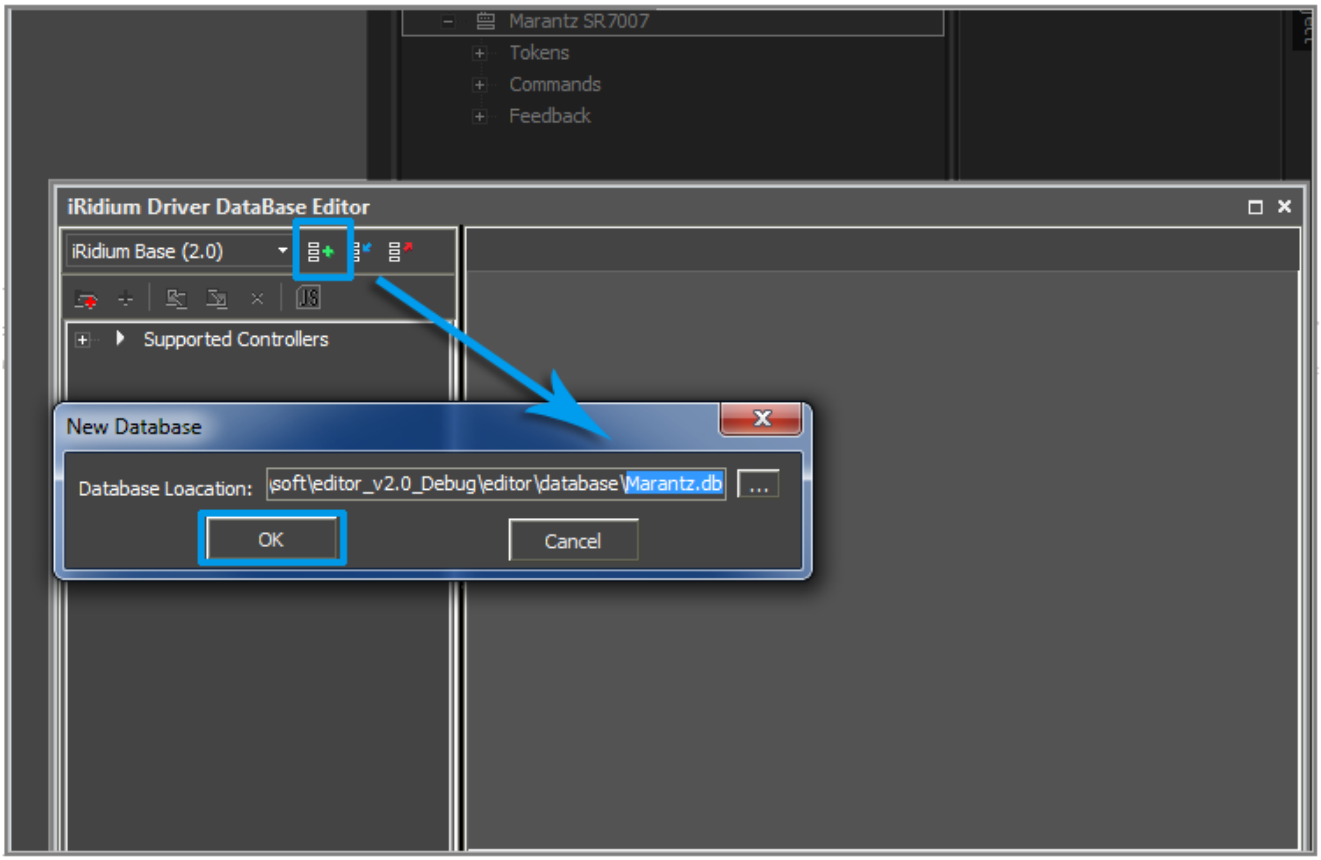
When the driver is ready for its use in other projects it is necessary to add it in **Device Base**:

- Open the **Device** tab
- Open **DB Editor**

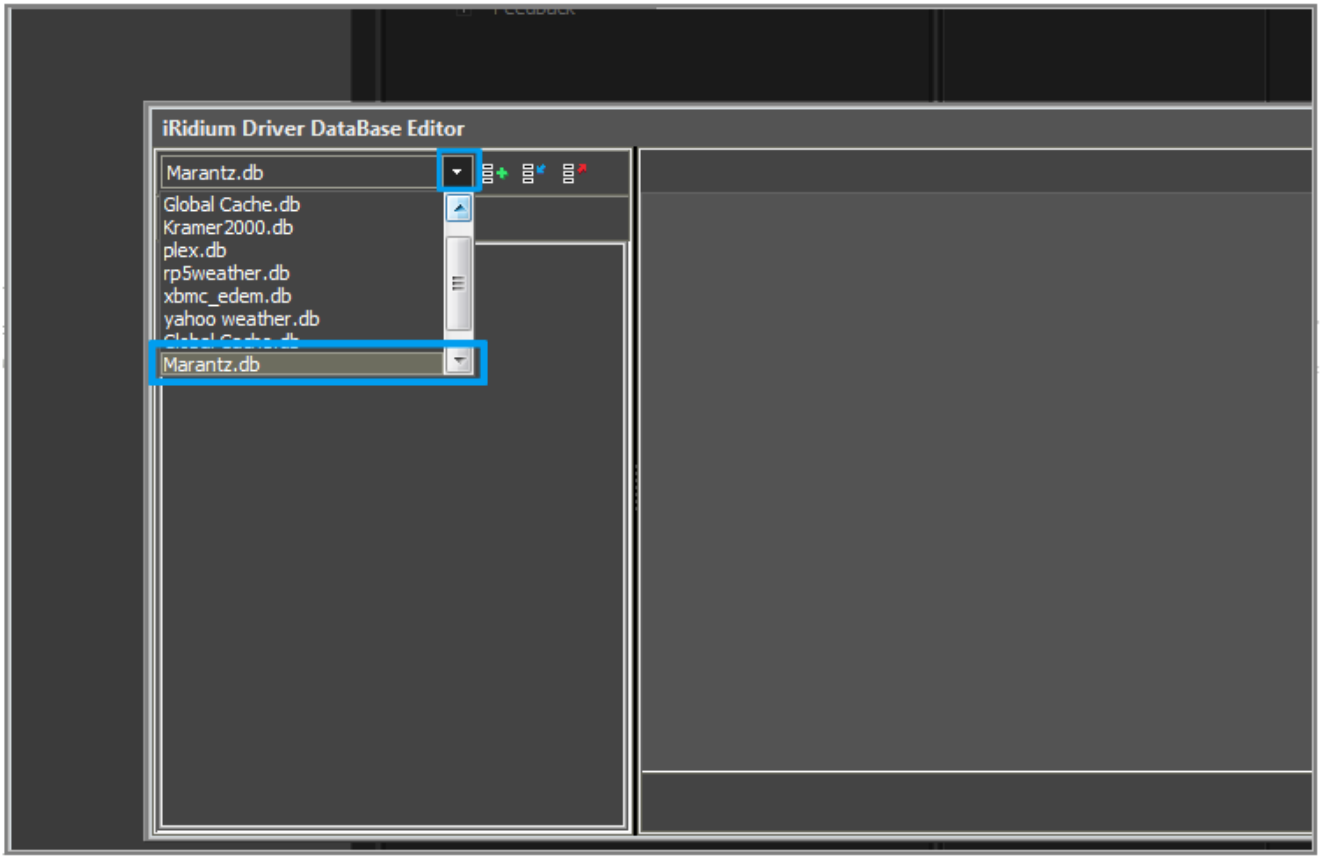



- Press on **New Database**
- Indicate the base name, in our example it is **Marantz.db**

Now your device base **Marantz** is created.

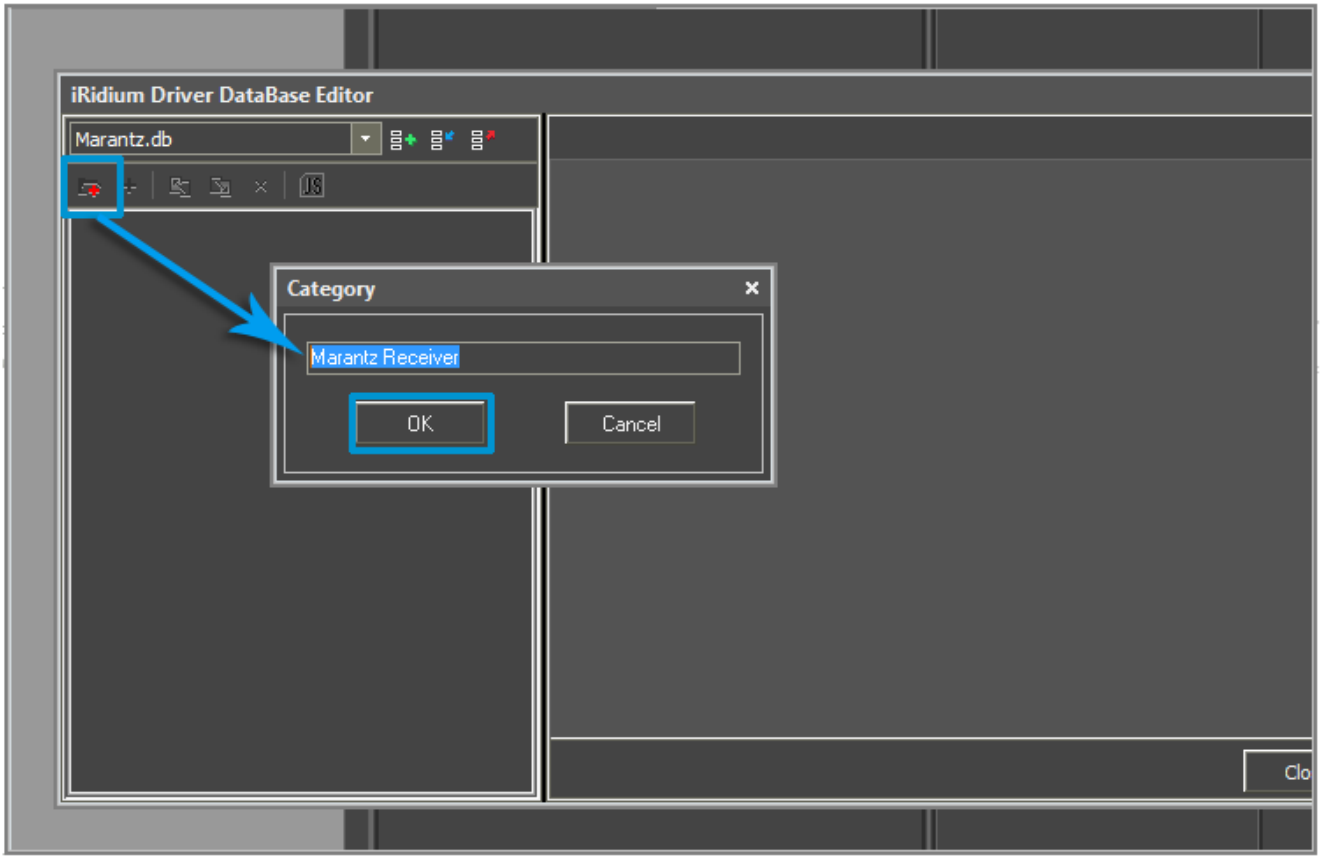



- Select your base from the list

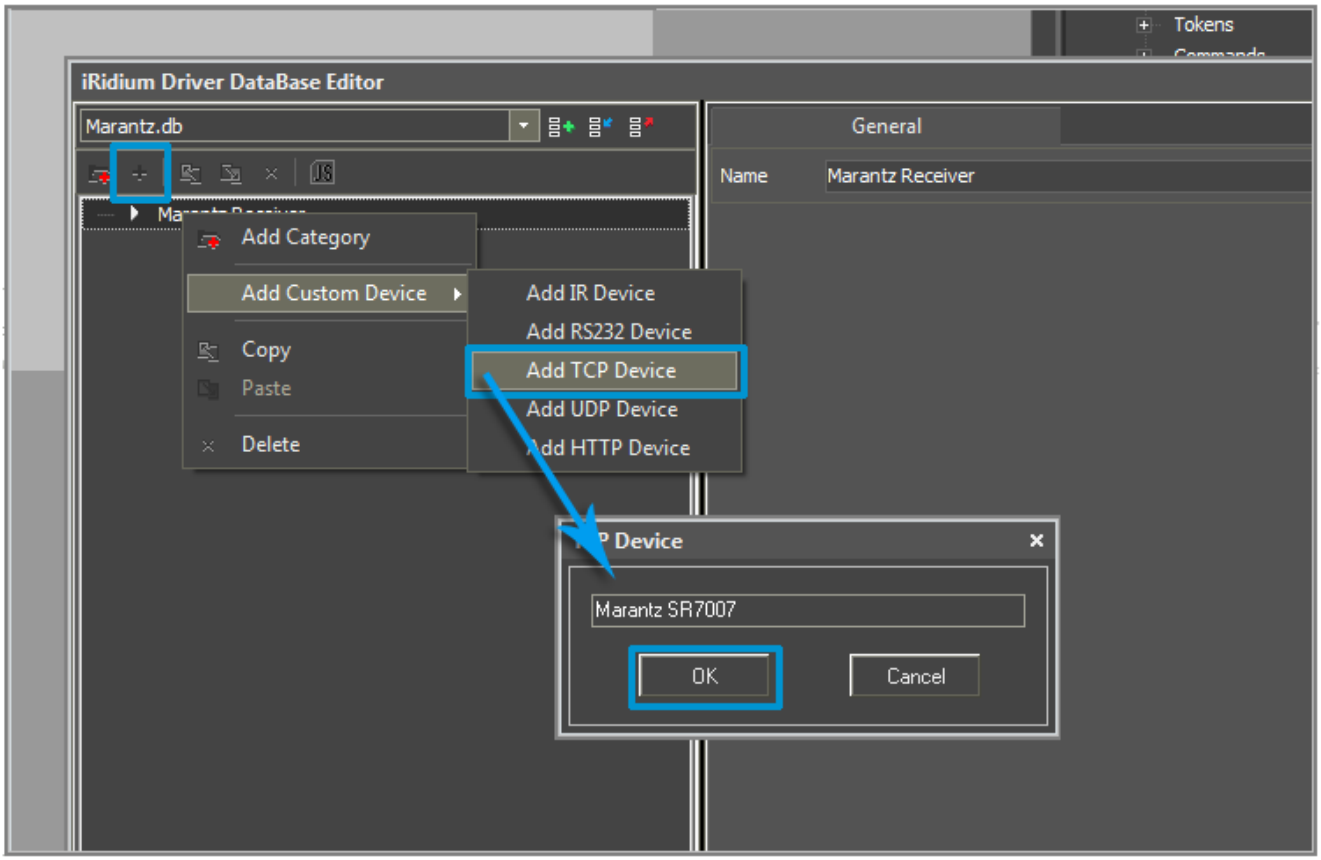



There are categories in *device base* to orientate easier, in our case **Marantz** is a reciever.

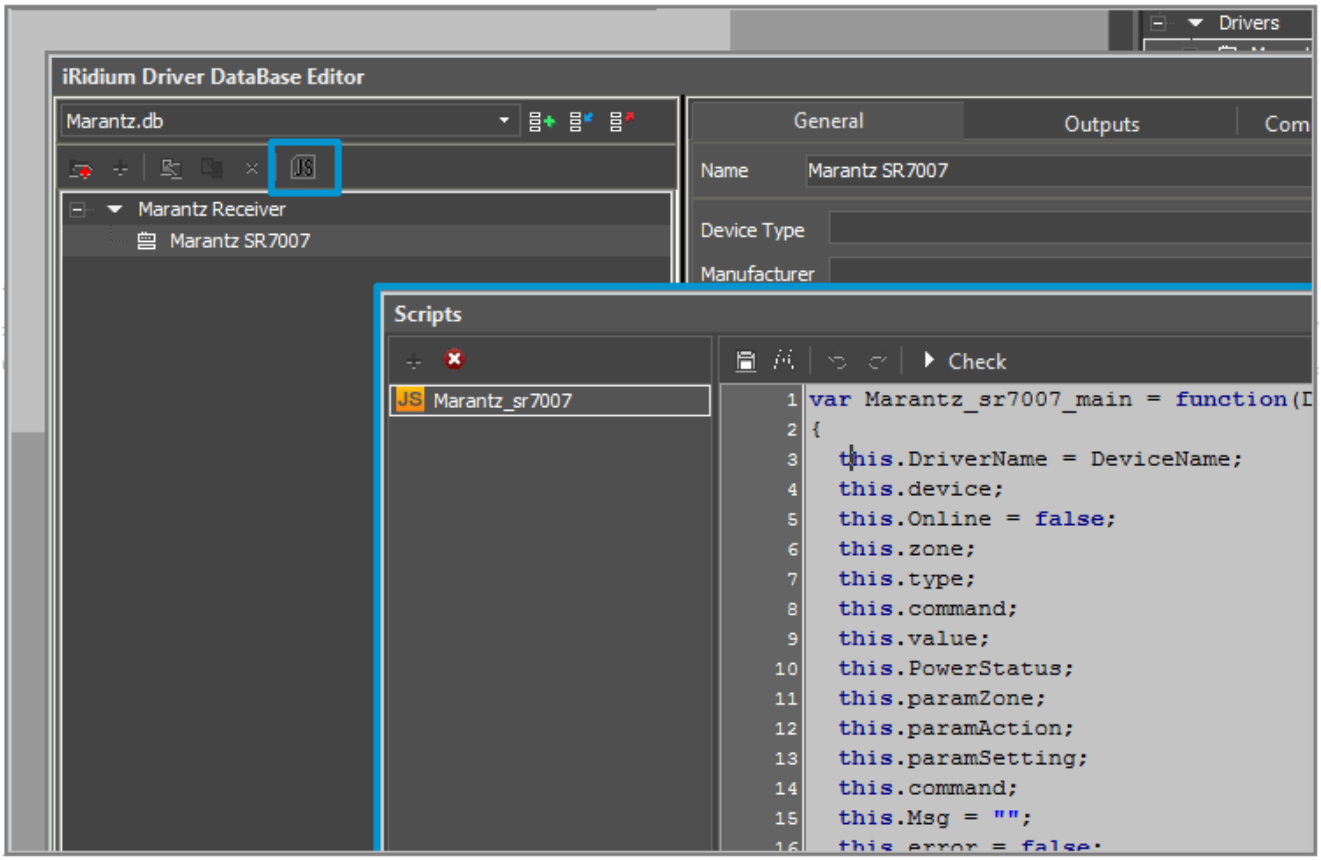
- Press on Add Category
- Indicate the name of the device category, in our case **Marantz Receiver**



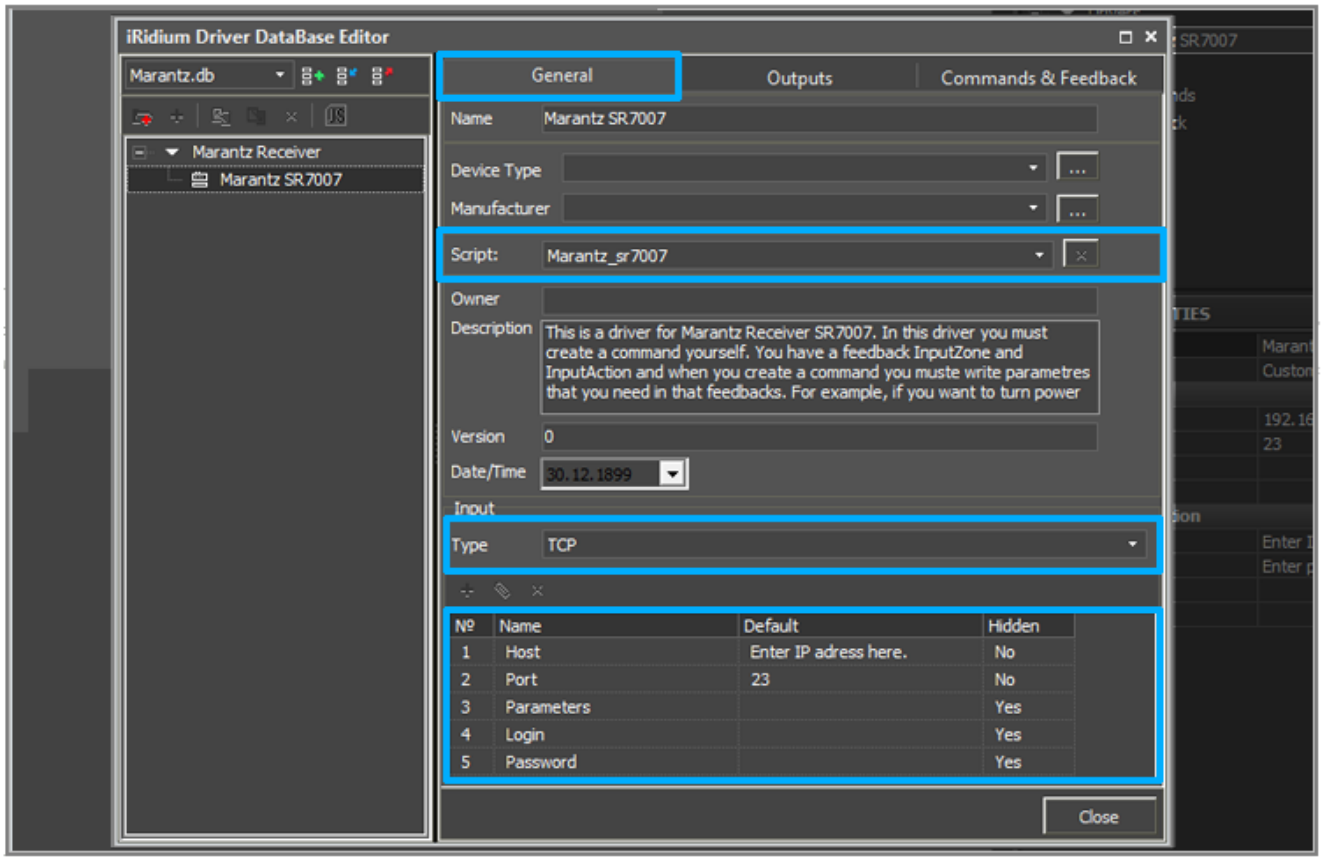

- Select the category
- Press on **Add Device**
- Select **Add Custom Device**
- Select the device type, in our case it is **TCP** - **Add TCP Device**
- Indicate the device name, in our case **Marantz SR7007**



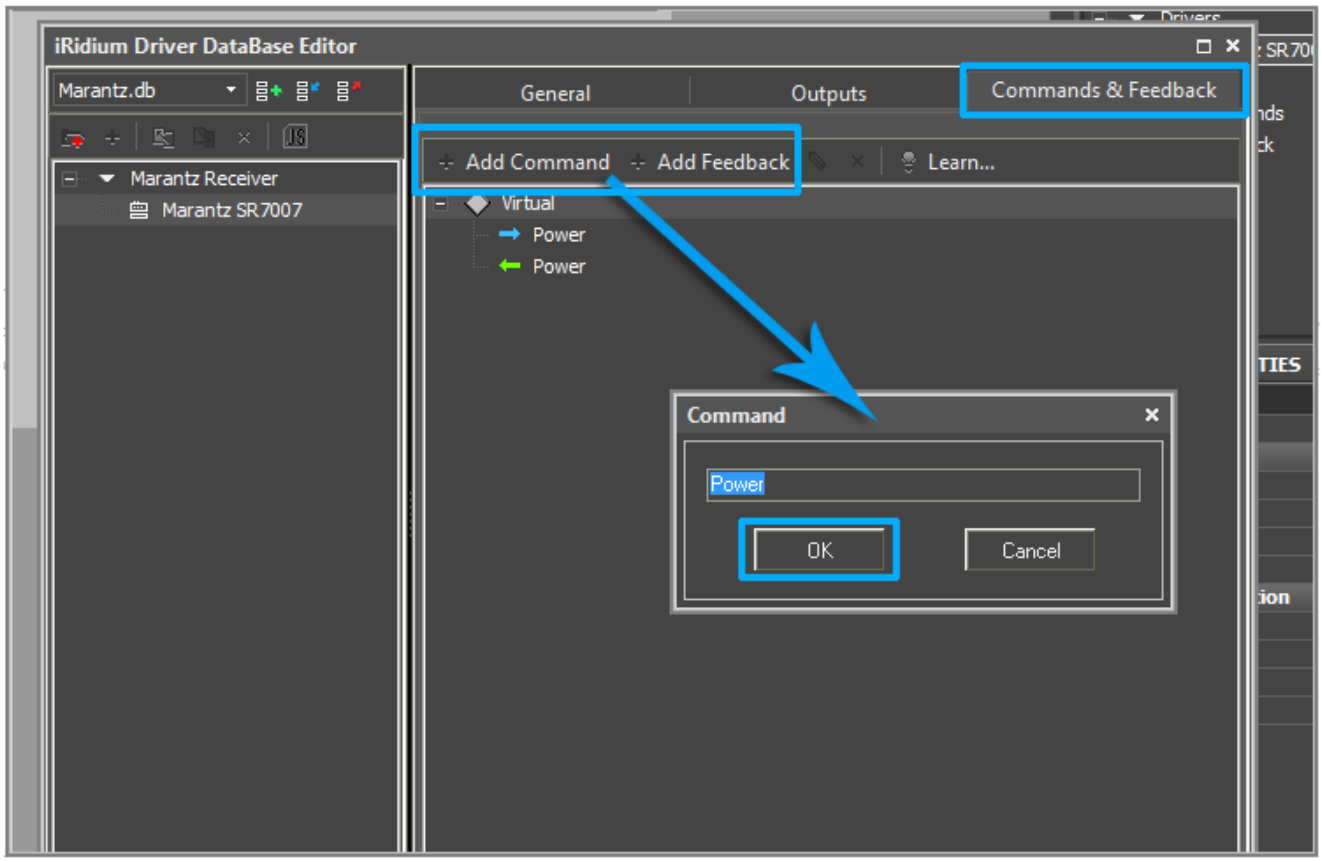

- Press on the JS button - **JS Editor**. For **DB Editor** (script editor of the global data base) there is a separate **JS Editor**, that is why when you open it your script will not be there.
- Create a new *script module* in it with the same name as in your project
- Copy your script of the driver in the new *module*




- Go to the **General** tab
- In the **Script** field select the module you added
- In the **Type** field select the type of input, in our case it is **TCP**
- The **Host** and **Port** fields can be set by default, **Marantz** works with port 23 by default




- Go to the **Commands & Feedback** tab
- Copy your commands **Commands** from the project
- Copy your **Feedbacks** from the project



Device outputs are indicated in the **Outputs** tab similarly to **Commands** and **Feedbacks**. At that adding of **the driver** in Device Base is completed.

# API for Working with Drivers

### Functions

| | |
|---|---|
| **IR.CreateDevice** | Creating a driver |
| **Connect** | Connecting to a device |
| **Disconnect** | Disconnecting from a device |
| **IR.GetDevice** | Referring to a device |
| **Set** | Setting up a value in the device channel |
| **Send** | Sending a command to a device |
| **InvokeAction** | Sending a command to an UPNP device |
| **Subscribe** | Subscribing to UPNP events |
| **UnSubscribe** | Unsubscribing from UPNP events |
| **HtmlDecode** | Substitution of reserved Html symbols |
| **JSON.Stringify** | Converting a JSON object to a string |
| **JSON.Parse** | Converting a string to a JSON object |
| **new XML** | Creating an XML object |
| **XML.ToString** | Converting an XML object to a string |

### Events

| | |
|---|---|
| **EVENT_RECEIVE_DATA** | Receiving data from a device in the bite format |
| **EVENT_RECEIVE_TEXT** | Receiving a string from a device |
| **EVENT_RECEIVE_EVENT** | Receiving an event (UPNP Event)from a device |
| **EVENT_ONLINE** | Connection with a device is established |
| **EVENT_OFFLINE** | Connection with a device is lost |
| **EVENT_TAG_CHANGE** | Changing a tag value |
| **EVENT_DEVICE_FOUND** | Finding an UPnP device |

# Download the example of the project