**iRidium API** are event-oriented scripts on the basis of Java Script.

With the help of the scripts you can complete many control tasks:

- working with graphic items
- working with sound
- working with drivers
- working with gestures
- working with lists
- working with system tags
- working with tokens

## Contents



## Main Concepts

- **Module** – a file with the js extension which is bound to the project.
- **Event** – an identifier of the process that taking place when the iRidium application is running.
- **Listener** – a function performed when the event assigned to it is activated.
- **IR** – a reserved word meaning that the command is the instruction of **iRidium API**.

```
  //Example of assigning Listener
  IR.AddListener(IR.EVENT_START, 0, function()
  // IR.EVENT_START – Event is activated at the application launch and the
following functions are performed
  {
        var test = 10;
        IR.Log(test); // Number 10 is displayed in the console
  });
```

The code is located in the modules. Listeners are located in the modules. Listeners receive events, objects and functions. Script support the structure of JSON and XML.

# Description of Uploading and Performing Scripts

The script can be launched only when a particular event is activated. Event types are presented in each API section.

One of the most important events is starting work of iRidium App, in **iRidium API** this event is written as **EVENT_START**. Developers recommend to initiate all functions required for operation of your project in this event.

When launching '*iRidium App* the whole code contained in the project is uploaded into the memory along with GUI.

The code stays in the memory while iRidium App is running. When closing iRidium App:

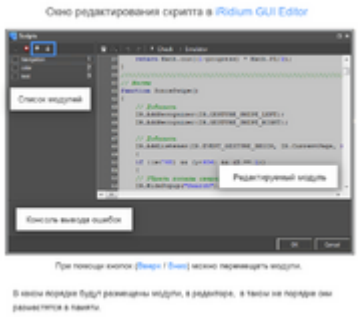
- **iOS** – performance of the code stops in 40 seconds
- **Android** - performance of the code doesn't stop (depending on the platform)
- **Windows** - performance of the code doesn't stop

All functions are performed in the order they are written.

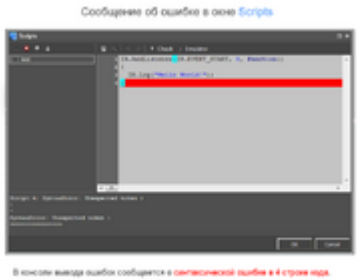
The code is performed asynchronously. If iRidium App refers to some device, iRidium App doesn't stop running the GUI and receives data in a separate stream as if you refer to the remote web site from your browser.

But when performing a big quantity of operations, for example, cycles, the whole GUI stops, waits until the operation is finished and after that continues its work.

# Working with Modules



The Scripts window



Error message

Writing scripts in **iRidium GUI Editor** is preformed in the **Scritps** window.

In the **Scritps** window you can do the following:

- create modules
- add modules from files
- delete modules
- move modules up or down

When performing each module is placed in the memory one by one.

The order of placing modules can be set in Editor with the help of up or down buttons in the left panel of the **Scripts** window.

All modules have their name space. If you plan to use several modules, you should think about the differences when setting names of variables and functions beforehand.

If in different project modules you activate the variable of function with the same identifier (name), then that the variable or function will be used which was placed in the memory last, i.e. in the module which is written later.

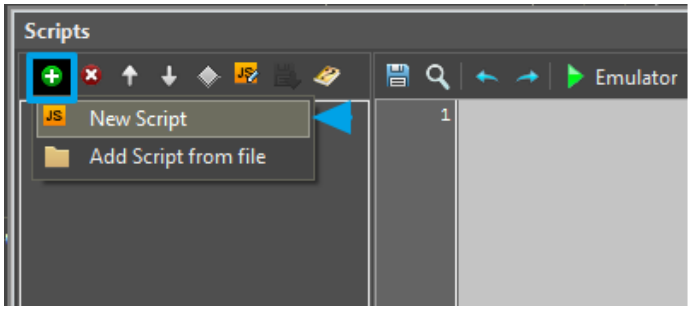
It does not refer to Listeners. If Listeners assigned to the same events they are performed together, one by one, without substituting each other. Developers do not recommend assigning Listeners to the same events inside one module as it creates illogical code structure.

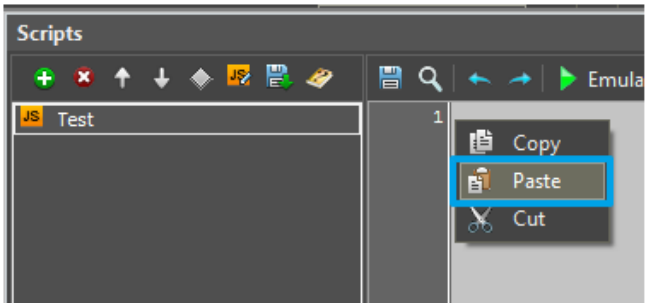
# How to Add Ready Scripts

There are several way of adding ready scripts in GUI Editor:

**Paste a script in a new module:**

1. Open the iRidium Script window 
2. Create a new script
3. Copy the script text from the source
4. Paste it in the created script ("Paste" in the right-click menu or Ctrl + V)



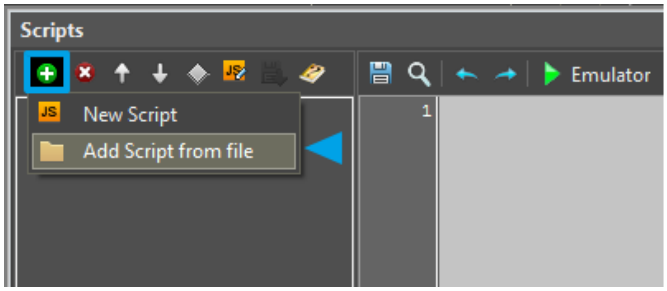



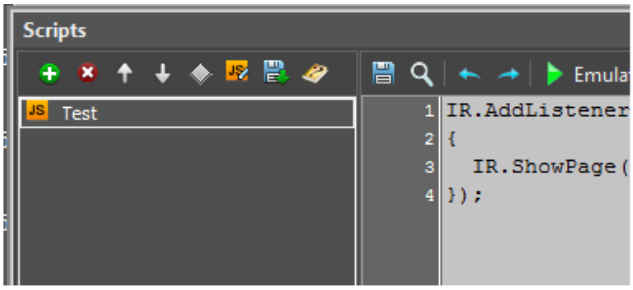

**Add a script as a file:**

If you have a ready script module *.js, it can be added in the iRidium Script window as follows:

1. Open the iRidium Script window 
2. Press on "Add Script from file"

3. Select the required script on your PC. Press on OK.




# Forming Command Chains

**Command chain** – the sequential activating of functions of outgoing result of the previous function.

Example of a command chain in **iRidium API**:

**IR.GetPage("Page 1").GetItem("Item 1").Width = 100;**

Each block highlighted in color is a chain link. Each link is separated from the other with the point - "."

After the IR command the Intellhelp system shows the list of available actions, such as:

- Activating of Listeners
- Referring to page/popup
- Referring to tokens or writing new values to tokens
- Creating drivers and changing their properties
- Defining actions with sound files
- Showing/hiding pages/popups

**GetPage("Page 1")** – the second chain link which serves to refer to the indicated page. After this command you can:

- Refer to an item on the indicated page/popup
- Change properties of the page/popup
- Count the number of items on the page
- Launch the macros assigned to the page

**GetItem("Item 1")** – the third chain link which serves to refer to the indicated item. After this link you can:

- Change properties of the indicated item
- Launch the macros assigned to the item
- Refer to the state of the item

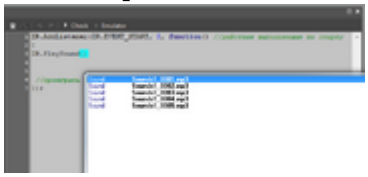
**Width** - the fourth chain link which serves to refer to properties of the indicated item. After this link you can:

- Set the required values of item properties

Commands available for each link are defined depending on the link selected before this link. Each next link gives the opportunity to manipulate the properties and methods of the previous link.

# IntellHelp

IntellHelp



Outputting local resources

The system of IntellHelp aims to simplify the process of creating and editing scripts in **iRidium GUI Editor**.

What does IntellHelp do?:

- Helps to build correct command chains.
- Defines which function you're writing at the moment and offers a list of available functions, methods and objects for it. It decreases the number of syntax mistakes and excludes the possibility of inputting incorrect data.
- Outputs the list of local resources.

IntellHelp appears automatically when starting the input but it can be activated any time by pressing **Ctrl + Space**.

It can be necessary if you forgot the command syntax, want to output the list of commands with similar syntax or the list of available local resources.

## Example of Outputting the List of Local Resources

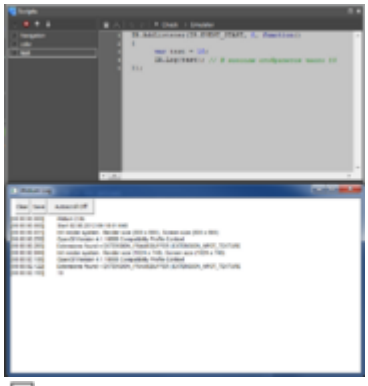
There are several sound files in project Gallery. When writing the **IR.PlaySound** command IntellHelp outputs the names of the available sound files.

```
IR.AddListener(IR.EVENT_START, 0, function() //Actions are performed at start
{
```

```
    IR.PlaySound("Sounds1_0003.mp3")
    //Play the sound file from Gallery with the name "Sounds1_0003.mp3"
});
```

# Debugging

In iRidium debugging of the written code is done with the help of embedded debugging tool – the output console **iRidium Log**.



The iRidium Log console

**iRidium Log** serves for:

- outputting any data in the debugging process
- automatic searching of errors and indicating the number of the string with the error
- saving the logging results in a file

You can find more information about how to output data in the console here

At the moment **iRidium Log** works on Windows OS only.

## Examples of Working with the iRidium Log Console

In the example below the variable value is output at the application launch:

```
  IR.AddListener(IR.EVENT_START, 0, function()
  {
        var test = 10;
        IR.Log(test); // Number  10 is displayed in the console
  });
```

In the example below data received from the device are output on the console:

```
  IR.AddListener(IR.EVENT_RECEIVE_TEXT, DEVICE, function(text)
  {
     IR.Log(text)
     // The “text“  variable containing all received data  is output in the
console.
  });
```

# Compatibility

**iRidium API** is fully compatible with Java Script. The exception is organizing a delay in Java Script. In **iRidium API** functions setTimeout and setInterval are not performed.

Organizing the delay in **iRidium API** can be done during the application work - the **EVENT_WORK** event is responsible for this identification of the process.

The **EVENT_WORK** event is activated continuously until the end of application work. It is performed with a particular cycle, the length of cycle depends on computational capabilities of the device. The code, which is written in the body of the function sent to Listener for this event, is performed nonstop until the work of the application is finished.

To organize the delay in iRidium at the **EVENT_WORK** event the function which is send to Listener receives the “time” variable. The “time” variable sends back the time (in ms) passed away from the last activation of **EVENT_WORK**.

## Example of Organizing the Delay in iRidium API

In the body of the function sent to Listener at the **EVENT_WORK** event you can increase the “time” number. As a result you receive the number of ms – this quantity can be compared with the required one, the time of the delay.

In the example below the following variable are set:

- **timer** – counter, it stores the times (in ms) which passes by
- **onTimer** – it stores the length of the delay (in ms)

The value of the counter increases until it equals the length of the delay..
When the delay time is over the "Hello World!" sign is output in the **iRidium Log** console and the counter is cleared. These actions are preformed while the application is running, in other words until activating the **EVENT_EXIT** event.

```
var timer = 0;
var onTime = 800; // 800 ms = 0.8 s
IR.AddListener(IR.EVENT_WORK, 0, function(time)
{
    timer += time;
    if(timer > onTime)
    {
        timer = 0;
        IR.Log("Hello World!");
    }
});
```