ads-tec GmbH

# IRF2000 IWL3000 SRC1000

# Application Note - ads-tec JSON RPC API specification

© ads-tec GmbH 2015

## Document History

| | | |
|---|---|---|
| 1.0 | 12/08/2013 | Initial Version |
| 1.1 | 03/09/2013 | Added objects gpio and network.ipsec.control |
| 1.2 | 10/01/2014 | Minor Changes |
| 1.3 | 31/03/2014 | Added notes about new internal “adstec.session” property |
| 1.4 | 04/07/2014 | Minor Changes |
| 2.0 | 28/10/2014 | New address |
| 2.1 | 06/11/2014 | Minor Changes and added SRC1000 specific gpio |
| 2.2 | 08/04/2015 | Updated of GPIO Methods for IRF2000 2.6.5 |

## Table of Contents

Introduction ........ 3
Overview ........ 4
1.1 session ........ 5
1.1.1 session.create() ........ 5
1.1.2 session.list() ........ 6
1.1.3 session.destroy() ........ 6
1.2 config ........ 7
1.2.1 config.get() ........ 7
1.2.2 config.table_get() ........ 8
1.2.3 config.sess_start() ........ 8
1.2.4 config.sess_abort() ........ 9
1.2.5 config.sess_commit() ........ 9
1.2.6 config.set() ........ 10
1.2.7 config.table_up () ........ 12
1.2.8 config.table_del() ........ 13
1.3 status ........ 14
1.3.1 status.get() ........ 14
1.4 gpio ........ 15
1.4.1 gpio.on(), gpio.off() ........ 16
1.4.2 gpio.list() ........ 17
1.4.3 gpio.get() ........ 18
1.4.4 gpio.get_pulses() ........ 19
1.5 network.ipsec.control ........ 20
1.5.1 network.ipsec.control.up() / down() ........ 20

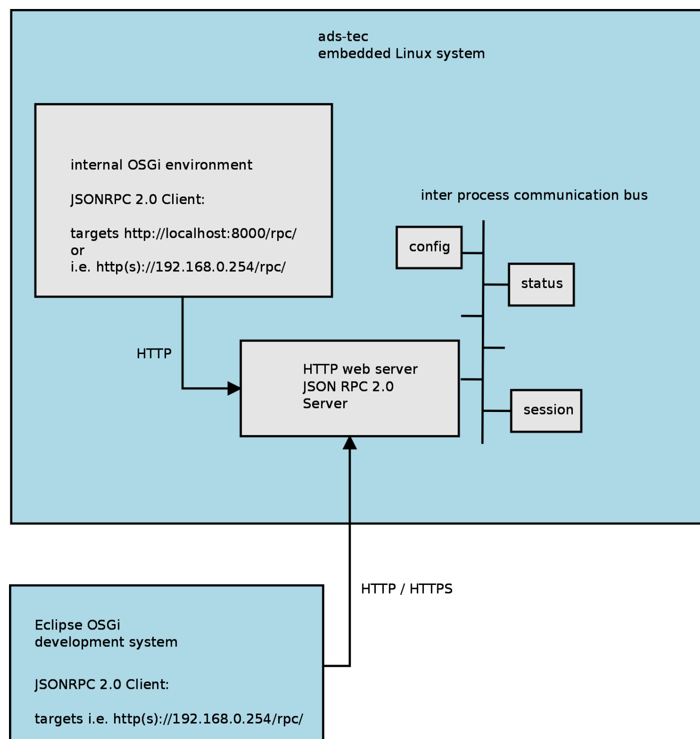
# Introduction

All ads-tec products of the IRF2000, IWL3000 or SRC1000 series provide an API communication with the device directly on target between the OSGi™ environment and the base system. The API is based on the JSON RPC 2.0 standard.

With this API it is possible:

- To access and change all configuration options.
- Gather current state information like IP addresses, 3G link state or whatever is of importance
- Control several device dependent systems like the battery control on the SRC1000, the wireless roaming control on the IWL3000 or the VPN and Big-LinX control on the IRF2000.

To understand how this API is working a rough overview is helpful.
All ads-tec embedded Linux systems contain an internal inter process communication bus. Most of the main internal systems connect to this bus to communicate with each other. From an external point of view this bus can be reached by the main HTTP web server. This main web server is not only serving the user interface on i.e. http://192.168.0.254/index.php but also has a special JSON RPC 2.0 programming interface at /rpc.




The web servers' JSON RPC interface at /rpc is reachable on the standard web server TCP port 80 or TCP port 443 if you want to use a secure connection with HTTPS. This is the main connection path for external programs like an Eclipse OSGi development system or any other custom tool which likes to make use of this API. From internal applications running on the OSGi system within the device a more direct path is available at a second small web server on localhost TCP port 8000. Here /rpc is the only available resource. Thus for a final application on the target http://localhost:8000/rpc is the main path to be used.

For external access authentication with a valid username and password is required. The special user "osgi" will be created on installing the osgi extension on the devices. Therefore you can use the user accounts "admin" or "osgi" to call the "session create" authentication method.
However if you run a bundle on the device it is not a good choice to hard code the user name and password inside the Java source code. As described in the Application Note "Apps with OSGi(tm) - 1st steps with the JSON RPC API" the system will provide you with a already allocated session id in the system environment which you can get by using the method bundleContext.getProperty("adstec.session").

# Overview

On the internal bus and therefore on the external /rpc interface there are three main objects common to all systems:

- config, provides methods for reading and change the configuration
- status, provides methods to gather state information
- session, the login and session module

Depending whether the product is of the IRF2000, IWL3000 or SRC1000 series other objects are available.

On the IRF2000 these are:

- network.ipsec.control

On IWL3000, IRF2000 and SRC1000:

- gpio

The basic structure of all JSON RPC requests always has the following structure:

- id, an id which is echoed in the reply to distinguish multiple requests at the same time
- jsonrpc, is always 2.0 and indicates the protocol version
- method, normally always „call“ if an object method is called. „list“ if you want to get a description of the object and its available methods.
- params, an JSON array, must always have the following four objects:
    - sid, the Session ID acquired by session create to authenticate the request
    - object, the object to call
    - method, the object’s method to call
    - JSON object with the parameters of the method or an empty object { } if no parameters are required

```
{
        "id" : "<user request id>",
        "jsonrpc": "2.0",
        "method" : "<call | list>",
        "params" : [
                "<sid>",
                "<object>",
                "<method>",
                { <object parameters> }
        ]
`
```

# 1.1 session

## 1.1.1 session.create()

```
"create": { "timeout": "Integer", "user": "String", "password": "String" }
```

The session create method takes three parameters:

- user: The username in the configuration system.
- password: The corresponding password from the configuration system.
- timeout (optional): session timeout, a default value of 600 seconds is used if the value is left empty.

The session create call is the only call which has the permission to get executed against the default sid 00000000000000000000000000000000. All other objects and methods need a valid sid.

Example request, login with admin/admin:

```
{
        "id" : "req-1",
        "jsonrpc": "2.0",
        "method" : "call",
        "params" : [
                "00000000000000000000000000000000",
                "session",
                "create",
                { "user": "admin", "password": "admin" }
        ]
`
```

Result:

```
{
          "jsonrpc": "2.0",
          "id": "1",
          "result": [
                    0,
                    {
                              "sid": "c945b35466f4ffc7c682a875a82d5be9",
                              "timeout": 600,
                              "expires": 600,
                              "acls": {
                                        "*": [
                                                  "*"
                                        ]
                              },
                              "data": {

                              }
                    }
          ]
}
```

The sid is a random string with 32 characters representing a hex value.
The acls objects consist of the access rights belonging to the session. The example above shows a wildcard access, in real examples you will get a list of allowed objects and methods depending on the user account and system type you're using.

Any access with an invalid session id will get an JSON error like this one:

```
{
          "jsonrpc": "2.0",
          "id": "1",
          "error": {
                    "code": -32002,
                    "message": "Access denied"
          }
}
```

### 1.1.2 session.list()

```
"list": {  }
```

Session list does not have any parameters and it will return all valid sessions. The result with one valid session is exactly the same as a on session.create(). If there are more valid sessions then the result array will simply have more objects attached.

### 1.1.3 session.destroy()

```
"destroy": { "sid": "String" }
```

Session.destroy() takes only one argument and will erase the specified session.

- sid: The Session ID to destroy

Request:

```
{
        "id" : "req-1",
        "jsonrpc": "2.0",
        "method" : "call",
        "params" : [
                "c945b35466f4ffc7c682a875a82d5be9",
                "session",
                "destroy",
                { "sid": " c945b35466f4ffc7c682a875a82d5be9" }
        ]
`
```

Result:

```
{
          "jsonrpc": "2.0",
          "id": "1",
          "result": [
                    0,
          ]
}
```

# 1.2 config

The internal configuration database consists of two different storage formats. The first are simple Strings of the form of var=value.
The other one are tables of Strings with columns IDs.

To alter the configuration a config session has to be acquired using sess_start(). This not the same as the RPC session ID but something complete different. All changes have to be set with the config session ID and are commited in one block using sess_commit()

```
'config':
        "get": { "keys": "Array" }
        "set": { "values": "Table", "cfg_session_id": "Integer" }
        "table_set": { "tablename": "String", "cfg_session_id": "Integer", "row": "Array" }
        "table_get": { "tablename": "String", "condition": "Table" }
        "table_del": { "tablename": "String",
                "condition": "Table", "cfg_session_id": "Integer" }
        "table_up": { "tablename": "String",
                "condition": "Table", "values": "Table", "cfg_session_id": "Integer" }
        "sess_start": {  }
        "sess_commit": { "cfg_session_id": "Integer" }
        "sess_abort": { "cfg_session_id": "Integer }
```

## 1.2.1 config.get()

```
"get": { "keys": "Array" }
```

config.get() takes an array of strings as parameter. Enter all configuration variables with values you like to know. For a list of variables valid on each product see the document “ads-tec-IT-Infrastructure-API-spec.pdf”

Example request, get IP address parameters:

```
{
        "id" : "req-1",
        "jsonrpc": "2.0",
        "method" : "call",
        "params" : [
                "c945b35466f4ffc7c682a875a82d5be9",
                "config",
                "get",
                { "keys": [ " lan_ipaddr", " lan_netmask"] }
        ]
`
```

Response:

```
{
          "jsonrpc": "2.0",
          "id": "1",
          "result": [
                    0,
                    {"lan_ipaddr": "192.168.0.254"},
                    {"lan_netmask": "255.255.255.0"}
          ]
}
```

Note: a non existing variable does not generate an error but returns an empty string!

### 1.2.2 config.table_get()

```
"table_get": { "tablename": "String", "condition": "Table" }
```

config.table_get() can be used to query an internal table like the user database. It takes two parameters.

- tablename: The name of the table
- condition: Query condition with up to two arguments

Example request (will query the users table for the “username” = “guest” and the column “enabled” = “0”):

```
{
        "id" : "req-1",
        "jsonrpc": "2.0",
        "method" : "call",
        "params" : [
                "c945b35466f4ffc7c682a875a82d5be9",
                "config",
                "table_get",
                { "tablename": "users", "condition": { "name": "guest", "enabled": "0" } }
        ]
`
```

Response:

```
{
          "jsonrpc": "2.0",
          "id": "1",
          "result": [
                    0,
                    {
                    "users": [{
                              "ROWID": "5",
                              "name": "guest",
                              "password_des": "qVV5o90eA3PYY",
                              "password_md5": "13ea3eb8d39ea87b0edde94e31903831",
                              "enabled": "0"
                              }]
          }]
}
```

### 1.2.3 config.sess_start()

```
"sess_start": {  }
```

sess_start() does not take any parameters and will return the next free config session ID. A valid config session ID is needed for any configuration change.

Example request, start a new session:

```
{
        "id" : "req-1",
        "jsonrpc": "2.0",
        "method" : "call",
        "params" : [
                "c945b35466f4ffc7c682a875a82d5be9",
                "config",
                "sess_start",
                { }
        ]
`
```

Response:

```
{
          "jsonrpc": "2.0",
          "id": "1",
          "result": [
                    0,
                    {
                              "cfg_session_id": 1
                    }
                    ]
}
```

### 1.2.4 config.sess_abort()

```
"sess_abort": { "cfg_session_id": "Integer }
```

sess_abort() takes a config session id as parameter. It will cancel all changes made with the specified ID.

Example request, abort session 1:

```
{
        "id" : "req-1",
        "jsonrpc": "2.0",
        "method" : "call",
        "params" : [
                "c945b35466f4ffc7c682a875a82d5be9",
                "config",
                "sess_abort",
                { "cfg_session_id": 1 }
        ]
`
```

Response:

```
{
          "jsonrpc": "2.0",
          "id": "1",
          "result": [
                  0,
          ]
}
```

### 1.2.5 config.sess_commit()

```
"sess_commit": { "cfg_session_id": "Integer" }
```

sess_commit() takes a config session ID as parameter. It will apply all changes made with the specified ID. This will result in the corresponding system changes and is similar to pressing the “apply” button in the system’s configuration web interface.

The sess_commit() will trigger internal dependencies checks which can generate an error if an invalid combination of parameters has been configured. For example it is not possible to enable a DHCP server on WAN if the operational mode is “Transparent Bridge” and thus the WAN interface is not available. In such a case the complete config set on the config session ID is discarded.

Example request, commit session 1:

```
{
        "id" : "req-1",
        "jsonrpc": "2.0",
        "method" : "call",
        "params" : [
                "c945b35466f4ffc7c682a875a82d5be9",
                "config",
                "sess_commit",
                { "cfg_session_id": 1 }
        ]
`
```

Response:

```
{
          "jsonrpc": "2.0",
          "id": "1",
          "result": [
                  0,
          ]
}
```

### 1.2.6 config.set()

```
"set": { "cfg_session_id": "Integer", "values": "Table", "verbose":"Bool"}
```

set() takes a config session ID which has to be acquired with sess_start() before. The other parameter is a table object with the values in the form: { “variable1”: “value”, “variable2”: “value2”, … }. The third parameter is optional and demands more verbose error messages when set to “True”.

The values will be checked against regular expressions within the system database, if a value is not permissible with the regular expression the whole set() call will be aborted and a corresponding error message will be returned. Without setting the verbose parameter only one error message per set() call is possible. Thus if you supply more than one wrong values only the first will be reported as being wrong and all following values will be discarded silently. Setting verbose to “True” will return a table containing error strings (in case at least one error occurred).

Example request, set system information parameters:

```
{
        "id": "req-1",
        "jsonrpc": "2.0",
        "params": [
                "c945b35466f4ffc7c682a875a82d5be9",
                "config",
                "set",
                { "cfg_session_id": 1,
                   "values": {
                        "system_contact":  "ads-tec",
                        "system_location": "oberaichen",
                   }
                }
        ]
}
```

Response:

```
{
          "jsonrpc": "2.0",
          "id": "1",
          "result": [
                    0,
          ]
}
```

Response with error, i.e, wrong IP address: "values": { "lan_ipaddr": "wrongip" }

```
{
          "jsonrpc": "2.0",
          "id": "1",
          "result": [
                    2,
                    {
                    "error": "IP address must be supplied or is invalid."
                    }
          ]
}
```

Response with error, i.e, wrong IP address: "values": { "lan_ipaddr": "wrongip" }

```
{
          "jsonrpc": "2.0",
          "id": "1",
          "result": [
                    2,
                    {
                    "error": { "lan_ipaddr":"IP address must be supplied or is invalid." }
                    }
          ]
}
```

## config.table_set()

```
"table_set": { "tablename": "String", "cfg_session_id": "Integer", "row": "Array" }
```

table_set() takes a config session ID, the table name to alter and a complete row array as parameter. The row array has to be complete with all columns belonging to the table. Like config.set() the system will check for regular expressions and can fail with an error.

Example request, add a new user:

```
{
        "id": "req-1",
        "jsonrpc": "2.0",
        "values":
        "params": [
                "c945b35466f4ffc7c682a875a82d5be9",
                "config",
                "table_set",
                { "cfg_session_id": 1,
                  "tablename": "users",
                  "row":
                    [
                              "newuser",
                              "",
                              "13ea3eb8d39ea87b0edde94e31903831",
                              "1"
                    ]
                }]
`
```

Response:

```
{
          "jsonrpc": "2.0",
          "id": "1",
          "result": [
                    0,
          ]
}
```

## 1.2.7 config.table_up ()

```
"table_up": { "tablename": "String",
               "condition": "Table", "values": "Table", "cfg_session_id": "Integer" }
```

table_up() takes a config session ID, the table name to alter and an update set as parameter “values”. Furthermore a condition has to be specified to elect the rows which shall be altered.

Example request, enable the guest user and set a new password:

```
{
        "id" : "req-1",
        "jsonrpc": "2.0",
        "method" : "call",
        "params" : [
                "c945b35466f4ffc7c682a875a82d5be9",
                "config",
                "table_up",
                {
                        "cfg_session_id": 1,
                        "tablename": "users",
                        "values":  {   "enabled": "1",
                                       "password_md5": "13ea3eb8d39ea87b0edde94e31903831"
                                   },
                        "condition": { "name": "guest" }
                }
        ]
```

Response:

```
{
          "jsonrpc": "2.0",
          "id": "1",
          "result": [
                    0,
          ]
}
```

Response with error, i.e. wrong column name „enable“ instead of „enabled“:

```
{
          "jsonrpc": "2.0",
          "id": "1",
          "result": [
                2,
                { "error":
                "one or both table columns do not exist: password_md5, enable on users"
                }
        ]
}
```

### 1.2.8 config.table_del()

```
"table_del": { "tablename": "String", "condition": "Table", "cfg_session_id": "Integer" }
```

table_del() takes a config session ID, the table name to alter and a condition to select the rows which shall be deleted.

Example request, delete the user “newuser” from the users table:

```
{
        "id" : "req-1",
        "jsonrpc": "2.0",
        "method" : "call",
        "params" : [
                "c945b35466f4ffc7c682a875a82d5be9",
                "config",
                "table_del",
                {
                        "cfg_session_id": 1,
                        "tablename": "users",
                        "condition": { "name": "newuser" }
                }
        ]
}
```

Response:

```
{
          "jsonrpc": "2.0",
          "id": "1",
          "result": [
                    0,
          ]
}
```

# 1.3 status

```
'status'
        "get": { "function": "String", "parameters": "Array" }
```

The status method can be used to query the system for all of its internal state values. For example all state information you find on the device web interface is catched with this method. Depending of the function called up to two parameters can be given to the function by using the parameters array

## 1.3.1 status.get()

Example request, get uptime and load:

```
{
        "id" : "req-1",
        "jsonrpc": "2.0",
        "method" : "call",
        "params" : [
                "c945b35466f4ffc7c682a875a82d5be9",
                "status",
                "get",
                {
                        "function": "uptime"
                        "parameters": [ "", "" ]
                }
        ]
}
```

Response:

```
{
          "jsonrpc": "2.0",
          "id": "1",
          "result": [
                    0,
                    {
                    "uptime": "10:05:48 up 1:51, load average: 0.00, 0.01, 0.00"
                    }
          ]
}
```

# 1.4 gpio

The gpio object is available on the IRF2000, IWL3000 and SRC1000 series devices. It provides manual control to the front side GPIO connectors and LEDs like CUT and ALARM on the IRF2000 or X1/X2 on the IWL3000. Further more the Reset Button on the front of the IRF2000 and IWL3000 devices can be polled.

```
'gpio'
        "on": { "signal": "String" }
        "off": { "signal": "String" }
        "get": { "signal": "String" }
        "get_pulses":{"signal":"String"}
        "list":{}
```

Valid signal identifiers for on/off:

- IRF2000/IWL3000 series devices
  - alarm or x1out
  - vpnup or x2out
- IRF2000 series devices
  - vpnled
  - cutled
- SRC1000 series devices
  - A2
  - A3
  - A4
  - A5

Valid signal identifiers for get:

- IRF2000/IWL3000 series devices
  - cut or x1in
  - vpnkey or x2in
  - button
- SRC1000 series devices
  - B2
  - B3
  - B4
  - B5

On the IRF2000 you should prevent conflicts with the GPIO and LED system. The system has to be switched to manual control before using the corresponding functions on VPN, ALARM or CUT!
Set the config variables to “disabled” or “enabled” to disable or enable the automatic control of the signals by the standard system:

gpio_cut_auto = disabled
gpio_alarm_auto = disabled
gpio_vpn_auto = disabled

On the SRC1000 the output pins (A2, A3, A4, A5) must be only be used if this output pin is dedicated for the customer.



adstec

### 1.4.1 gpio.on(), gpio.off()

Example request, enable the alarm output:

```
{
        "id" : "req-1",
        "jsonrpc": "2.0",
        "method" : "call",
        "params" : [
                "c945b35466f4ffc7c682a875a82d5be9",
                "gpio",
                "on",
                {
                        "signal": "alarm"
                }
        ]
}
```

Response:

```
{
          "jsonrpc": "2.0",
          "id": "1",
          "result": [
                    0,
                    {
                    "button": "on"
                    }

          ]
}
```

### 1.4.2 gpio.list()

(available starting from IRF2000 2.6.5 and SRC1000 1.2.7)
Example request, list all available gpio signals.

```
{
	"id" : "req-1",
	"jsonrpc": "2.0",
	"method" : "call",
	"params" : [
		"c945b35466f4ffc7c682a875a82d5be9",
		"gpio",
		"list",
		{
		}
	]
}
```

Response, example from IRF2000:

```
{
	"jsonrpc": "2.0",
	"id": "1",
	"result": [
		0,
		{
		"signals": [
			{
				"signal": "cut",
				"direction": "in"
			},
			{
				"signal": "vpnkey",
				"direction": "in"
			},
			{
				"signal": "alarm",
				"direction": "out"
			},
			{
				"signal": "vpnup",
				"direction": "out"
			},
			{
				"signal": "vpnled",
				"direction": "out"
			},
			{
				"signal": "cutled",
				"direction": "out"
			},
			{
				"signal": "button",
				"direction": "in"
			}
			]
		}
	]
}
```

### 1.4.3 gpio.get()

Example request, get the button state:

```
{
        "id" : "req-1",
        "jsonrpc": "2.0",
        "method" : "call",
        "params" : [
                "c945b35466f4ffc7c682a875a82d5be9",
                "gpio",
                "get",
                {
                        "signal": "button"
                }
        ]
}
```

Response:

```
{
          "jsonrpc": "2.0",
          "id": "1",
          "result": [
                    0,
                    {
                    "button": "on"
                    }
          ]
}
```

### 1.4.4 gpio.get_pulses()

(available starting from IRF2000 2.6.5 and SRC1000 1.2.7)
The pulse counter will count all raising and falling edges. Thus if you press the button once you will get a pulse count of 2.
The sampling interval is determined by the sum of the system variables ems_in_time, ems_logic_time, ems_out_time. The default values are 90 000 us, 20 000 us, 90 000 us = 200 000 us. They can be decreased down to 1000 us, thus resulting in a sampling interval of 3ms.

*Note for SRC1000 systems:*
*Depending on the system performance and other enabled plugins in the ems controller the fastest possible sampling interval is highly depending on other factors like CPU load and external communication.*

Example request, get the button state:

```
{
        "id" : "req-1",
        "jsonrpc": "2.0",
        "method" : "call",
        "params" : [
                "c945b35466f4ffc7c682a875a82d5be9",
                "gpio",
                "get_pulses",
                {
                        "signal": "button"
                }
        ]
}
```

Response:

```
{
          "jsonrpc": "2.0",
          "id": "1",
          "result": [
                    0,
                    {
                    "button": 2
                    }
          ]
}
```

# 1.5 network.ipsec.control

## 1.5.1 network.ipsec.control.up() / down()

The ipsec.control allows switching on/off existing IPsec policies. The IPsec policies must be configured as Active (Switched) in the configuration interface.

```
'network.ipsec.control'
        "up": { "id": "Integer" }
        "down": { "id": "Integer" }
```

Example request, activate first row in IPsec connection:

```
{
        "id" : "req-1",
        "jsonrpc": "2.0",
        "method" : "call",
        "params" : [
                "c945b35466f4ffc7c682a875a82d5be9",
                "network.ipsec.control",
                "up",
                {
                        "id": "o"
                }
        ]
}
```

Response:

```
{
          "jsonrpc": "2.0",
          "id": "1",
          "result": [
                    0,
          ]
}
```

-

*OSGi™ is a trademark or a registered trademark of the OSGi Alliance in the United States, other countries, or both*