

ads-tec IIT GmbH

IRF2000 IRF3000 IRF1000

**Application Note - ads-tec JSON
RPC API specification**



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
2.3	19/02/2016	Added statusled as GPIO
3.0	16/02/2021	Reduced to IRF1000 and IRF2000, removed all OSGi related topics.
3.1	20/03/2021	Added new Big-LinX "statusd" object
3.2	28/09/2021	Added new "file" object and new methods: "config_import()", "config_get_values()", "get_defaults", "table_schema" and "export_pages()" on the "config" object.
3.3	23/09/2022	Added IRF3000 and new Big-LinX IIOT Push Object: blxpush
3.4	18/11/2022	Updated new Big-LinX IIOT Push Object: blxpush
3.5	11/09/2024	Fixed parameter name for certificate upload

Table of Contents

Introduction.....	4
Overview	5
1.1 Session	6
.....	6
1.1.1 session.create().....	6
1.1.2 session.list().....	7
1.1.3 session.destroy().....	7
1.2 config.....	8
1.2.1 config.get().....	8
1.2.2 config.get_values().....	9
1.2.3 config.get_default()	10
1.2.4 config.export_pages()	11
1.2.5 config.import_config().....	12
1.2.6 config.table_get().....	14
1.2.7 config.sess_start().....	15
.....	16
1.2.8 config.sess_abort().....	17
1.2.9 config.sess_commit()	17
1.2.10 config.set().....	19
1.2.11 config.table_up ().....	22
1.2.12 config.table_del().....	23
1.3 status	24
1.3.1 status.get()	24
1.4 gpio	25
1.4.1 gpio.on(), gpio.off().....	26
1.4.2 gpio.list()	27
1.4.3 gpio.get().....	29

Application Note - ads-tec JSON RPC API specification 3.5

1.4.4	gpio.get_pulses()	30
1.5	network.ipsec.control	31
1.5.1	network.ipsec.control.up() / down()	31
1.6	statusd	32
1.6.1	statusd.blx_status().....	32
1.6.2	statusd.blx_vpn_up()	33
1.6.3	statusd.vpn_down().....	33
1.7	file.....	34
1.7.1	file.write().....	34
1.7.1.1	Settings file Upload.....	34
1.7.1.2	Certificate file Upload	35
1.7.2	file.read().....	37
1.7.2.1	Settings file Download	37

Introduction

All ads-tec products of the IRF2000 and IRF1000 series provide an API communication with the device. 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 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.

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
- file, provides a mechanism to upload base64 encoded binary files like certificates or setting files.

On the IRF1000 to IRF3000 these objects are available too:

- network.ipsec.control
- gpio, provides interaction with LEDs, buttons and external I/O pins
- statusd, provides all Big-LinX controls
- blxpush, provide data transfer for Big-LinX IIOT dashboards

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

```
'session':
  "create": { "timeout": "Integer", "user": "String", "password": "String" }
  "list": { }
  "destroy": { "sid": "String" }
```

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": "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,
      "acIs": {
        "*": [
          "*"
        ]
      },
      "data": {
      }
    }
  ]
}
```

The sid is a random string with 32 characters representing a hex value.

Application Note - ads-tec JSON RPC API specification 3.5

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 committed in one block using sess_commit().

The only exception is the config_import() method which is a bulk import and contains the config session in itself.

```
'config':
  "get": { "keys": "Array" }
  "get_values": { "keys": "Array" }
  "set": { "values": "Table", "cfg_session_id": "Integer" }
  "get_default": { "keys": "Array" }
  "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" }
  "table_schema": { "tablename": "String" }
  "export_pages": { "pages": "Array" }
  "import_config": { "jsondata": "Table" }
  "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.get_values()

```
"get_values": { "keys": "Array" }
```

config.get_values() takes an array of strings as parameter. Enter all configuration variables with values you like to know. The response will contain the default value, read/write permission for the running session and JSON interested value and the raw string value.

Example request, get IP address parameters:

```
{
  "id": "req-1",
  "jsonrpc": "2.0",
  "method": "call",
  "params": [
    "c945b35466f4ffc7c682a875a82d5be9",
    "config",
    "get_values",
    {
      "keys": [
        "wan_status",
        "year"
      ]
    }
  ]
}
```

1.2.3 config.get_default()

```
"get_values": { "keys": "Array" }
```

config.get_default() takes an array of strings as parameter. Enter all configuration variables with values you like to know. The response will contain the default value of the currently running firmware and product.

Example request, get IP address parameters:

```
{
  "id": "req-1",
  "jsonrpc": "2.0",
  "method": "call",
  "params": [
    "c945b35466f4ffc7c682a875a82d5be9",
    "config",
    "get_default",
    {
      "keys": [
        "wan_status",
        "year"
      ]
    }
  ]
}
```

Response:

```
{
  "jsonrpc": "2.0",
  "id": "req-1",
  "result": [
    0,
    {
      "result": [
        {
          "wan_status": "enabled"
        },
        {
          "year": "2021"
        }
      ]
    }
  ]
}
```

1.2.4 config.export_pages()

```
"export_pages": { "pages": "Array" }
```

config.export_pages() takes an array of page ID strings as arguments. It will return a JSON table with all containing settings of the given pages. This export can be imported using the config_import() method.

Example request

```
{
  "id": "req-1",
  "jsonrpc": "2.0",
  "method": "call",
  "params": [
    "be9a45cca36a0909e390c8c9e10ee297",
    "config",
    "export_pages",
    {
      "pages": [
        "SYSTEMINFO",
        "IOT_DATASETS",
        "IOT_METRICS"
      ]
    }
  ]
}
```

Response:

```

{
  "jsonrpc": "2.0",
  "id": "req-1",
  "result": [
    0,
    {
      "configdata": {
        "contact_email": "mailbox@ads-tec.de",
        "contact_name": "ads-tec",
        "contact_phone": "0049702225220",
        "system_location": "Nuertingen",
        "system_name": "IRF1421",
        "systemname_sndyn": "enabled"
      },
      "tableinsert": [
        {
          "tablename": "iot_datasets",
          "data": [
            {
              "ROWID": "1",
              "dataset_name": "Measure",
              "description": "created by wizard",
              "iot_datasets_group_id": "1"
            }
          ]
        },
        {
          "tablename": "iot_metrics",
          "data": [
            {
              "ROWID": "1",
              "storage_int": "1",
              "retension": "1",
              "name": "T",
              "dataset_id": "1",
              "description": "generated by wizard"
            }
          ]
        }
      ]
    }
  ]
}

```

1.2.5 config.import_config()

```
"import_config": {"jsondata": "Table"}
```

import_config() takes a complex JSON data structure as argument. See config.export_pages() output as example. There are three main sections.

- configdata: table of all flat config parameters which shall be set.
- tableinsert: JSON structure which contains a direct mapping of SQL table content of the internal configuration database.
- tableupdate: JSON structure of SQL like Update statements
- tabeldelete: JSON structure of SQL like Delete statements

Application Note - ads-tec JSON RPC API specification 3.5

The `import_config()` method is a bulk import. Thus it does not need a configuration session id. The import data will be parsed and checked for obvious logical problems. It will abort on the first parse error, write permission or regular expression error. Nothing at all will be altered if such an error happens.

Please note that the logical interdependencies of the configuration parameters are quite complex, and the integrated checks cannot be perfect. Great care must be taken if a custom build JSON data structure is to be imported.

The best example is the output of the `export_pages()` call. However, the `tableupdate` and the `tabledelete` section will not appear there. Therefore, we go in details on these two here:

tabledelete:

Example request, delete the "guest" and the "testuser" user in the users table:

```
{
  "id": "req-1",
  "jsonrpc": "2.0",
  "method": "call",
  "params": [
    "8ba8b5ab1dfe02c3d9cbc195493ea136",
    "config",
    "import_config",
    {
      "jsondata": {
        "tabledelete": [
          {
            "tablename": "users",
            "data": [
              {
                "name": "guest"
              },
              {
                "name": "testuser"
              }
            ]
          }
        ]
      }
    }
  ]
}
```

Response on success:

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

tableupdate:

Application Note - ads-tec JSON RPC API specification 3.5

Example request, alter the "users" table and set the "enabled" row to value "1". Two key/value pairs are required. The first is the key and value to be set and the second is to select the rows where the given key matches the given value.

```
{
  "id": "req-1",
  "jsonrpc": "2.0",
  "method": "call",
  "params": [
    "8ba8b5ab1dfe02c3d9cbc195493ea136",
    "config",
    "import_config",
    {
      "jsondata": {
        "tableupdate": [
          {
            "tablename": "users",
            "data": [
              {
                "enabled": "1", "name": "guest"
              }
            ]
          }
        ]
      }
    }
  ]
}
```

Response on success:

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

1.2.6 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"
        }
      ]
    }
  ]
}

```

1.2.7 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
    }
  ]
}
```


1.2.8 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
  ]
}
```

1.2.9 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
  ]
}
```

1.2.10 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."
      }
    }
  ]
}
```

Application Note - ads-tec JSON RPC API specification 3.5

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",
  "params": [
    "c945b35466f4ffc7c682a875a82d5be9",
    "config",
    "table_set",
    {
      "cfg_session_id": 1,
      "tablename": "users",
      "row": [
        "newuser",
        "",
        "13ea3eb8d39ea87b0edde94e31903831",
        "1"
      ]
    }
  ]
}
```

Response:

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

1.2.11 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"
    }
  ]
}
```

1.2.12 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 caught 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 and IRF100 series devices. It provides manual control to the front side GPIO connectors and LEDs like CUT and ALARM on the IRF2000. Further more the Reset Button on the front of the IRF2000 and IRF1000 devices can be polled.

Depending of the firmware version other “signals” are available. The currently available signals can be polled using the gpio.list() method.

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

For example, valid signal identifiers for on/off:

- IRF2000 series devices
 - alarm or x1out
 - vpnup or x2out
- IRF2000 series devices
 - vpnled
 - cutled
 - statusled

Valid signal identifiers for get:

- IRF2000 series devices
 - cut or x1in
 - vpnkey or x2in
 - button

On the IRF2000/IRF1000 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
```

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 IRF1000 1.0.0)

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.

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  
  ]  
}
```

1.6 statusd

The "statusd" object allows the control of the Big-LinX VPN. There are three methods available.

```
'statusd'
  "blx_status":{}
  "blx_vpn_up":{}
  "blx_vpn_down":{}
```

1.6.1 statusd.blx_status()

Example request, get blx_status():

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

Response:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": [
    0,
    {
      "blx_status": {
        "cardstate": "VPNSC_CS_READY",
        "tokenlabel": "YRVYRUUSMGHZ",
        "openvpnsctanstate": "VPNSC_OPENVPNSCAN_READY",
        "vpn_state_name": "",
        "vpn_oldstate_name": "",
        "vpn_state_desc": "",
        "vpn_ip": "",
        "vpn_permanent": 0,
        "vpn_server_ip": "",
        "pintries": 1,
        "savepin": "",
        "vpn_ctrl_state": 0,
        "wwh_service": "enabled"
      }
    }
  ]
}
```

The fields within the resulting table have the following meanings:

- cardstate: State String of the internal state machine which will search detect and initialize the crypto smartcard or software certificate. VPNSC_CS_READY means that the VPN can be activated
- tokenlabel: Common Name of the Big-LinX X.509 VPN certificate which is either on the crypto smartcard or as software credential. Depending of the generation of certificate this is either a readable vendor string with number or a UUID style identifier as in the example above.
- openvpnsctanstate: State String of the second internal state machine regarding the OpenVPN process management. (for internal use)

Application Note - ads-tec JSON RPC API specification 3.5

- `vpn_state_name`: State String of the OpenVPN process or empty if the process is currently not running.
- `vpn_state_name_old`: State String of the previous state of the OpenVPN process
- `vpn_state_desc`: (deprecated field, please do not use)
- `vpn_ip`: IP address of the virtual VPN network interface if the VPN channel is up, or empty string otherwise
- `vpn_server_ip`: Empty string in case the VPN channel is down. String containing the public IPv4 address of the chosen VPN server, TCP Port, and local IP address of the uplink interface.
- `pintries`: (deprecated field, please do not use, always 1 for compatibility)
- `savepin`: (deprecated field, please do not use)
- `vpn_ctrl_state`: Target state von the VPN channel. Either 1 if the VPN shall be up or 0 if the VPN shall be down. This field will directly mirror the result of the `blx_vpn_up()/down()` methods. In case of an active acknowledge configuration, the value will be 1 while the system is waiting for acknowledgement.
- `wwh_service`: configured state of the WWH service, should always be "enabled" (for internal use)

1.6.2 `statusd.blx_vpn_up()`

Example request, start the Big-LinX VPN.

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

Response:

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

1.6.3 `statusd.vpn_down()`

Example request, stop the Big-LinX VPN.

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

Response:

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

1.7 file

The "file" object allows the control of the Big-LinX VPN. There are three methods available.

```
'file'
  "read":{"path":"String"}
  "write":{"path":"String","data":"String",
    "append":"Boolean","mode":"Integer"}
  "list":{"path":"String"}
  "stat":{"path":"String"}
  "md5":{"path":"String"}
  "remove":{"path":"String"}
```

1.7.1 file.write()

The write() method can only take two valid paths as upload parameter. In case of a cf2 file it must be uploaded to /tmp/upsettings and in case of a certificate it must be uploaded to /tmp/upcerts. No other paths are allowed.

1.7.1.1 Settings file Upload

The upload of a cf2 settings file contains of two steps. First the file as it is must be uploaded via the file.write() methods. Afterwards the system must be triggered to parse this file by using the config.set() method.

Example request, first upload a cf2 file:

```
{
  "id": "req-1",
  "jsonrpc": "2.0",
  "method": "call",
  "params": [
    "c945b35466f4ffc7c682a875a82d5be9",
    "file",
    "write",
    {
      "data": "<base64 encoded file data>",
      "append": False,
      "mode": 0o700,
      "path": "/tmp/upsettings/savedsettings.cf2",
    }
  ]
}
```

Second trigger the cf2 file parser using a sequence of config.sess_start(), config.set() and config.sess_commit().

The config.set() of this sequence for example:

```

{
  "id": "req-1",
  "jsonrpc": "2.0",
  "params": [
    "c945b35466f4ffc7c682a875a82d5be9",
    "config",
    "set",
    {
      "cfg_session_id": 1,
      "values": {
        "restore_settings_now": "123",
        "restore_settings_filename": "savedsettings.cf2",
        "restore_settings_sha256":
"a8963b2de5710708fe3a200f3340d48fe2e09f64648801c865f82ec3f3bdb8f6",
      }
    }
  ]
}

```

The `restore_settings_now` variable is a trigger variable, the content does not matter and will be discarded by the system. The `restore_settings_sha256` value must contain the SHA256 checksum of the previously uploaded file.

1.7.1.2 Certificate file Upload

The upload of a certificate file two steps have to be undertaken. First the file as it is must be uploaded via the `file.write()` methods. Afterwards the system must be triggered to parse this file by using the `config.set()` method.

Example request, first upload a certificate file:

```

{
  "id": "req-1",
  "jsonrpc": "2.0",
  "method": "call",
  "params": [
    "c945b35466f4ffc7c682a875a82d5be9",
    "file",
    "write",
    {
      "data": "<base64 encoded file data>",
      "append": False,
      "mode": 0o700,
      "path": "/tmp/upcerts/mycert.p12",
    }
  ]
}

```

Second trigger the cf2 file parser using a sequence of `config.sess_start()`, `config.set()` and `config.sess_commit()`.

The `config.set()` of this sequence for example:

```
{
  "id": "req-1",
  "jsonrpc": "2.0",
  "params": [
    "c945b35466f4ffc7c682a875a82d5be9",
    "config",
    "set",
    {
      "cfg_session_id": 1,
      "values": {
        "filename_password": "myp12passphrase",
        "upload_certfile_now": "123",
        "upload_cert_filename": "mycert.p12",
        "upload_cert_file_sha256":
          "a8963b2de5710708fe3a200f3340d48fe2e09f64648801c865f82ec3f3bdb8f6",
      }
    }
  ]
}
```

The `upload_cert_file_now` variable is a trigger variable, the content does not matter and will be discarded by the system. The `upload_cert_file_sha256` value must contain the SHA256 checksum of the previously uploaded file. The parameter "filename_password" must only be added if the cert file is protected by passphrase.

1.7.2 file.read()

The read() method can only be used to download a settings file. The settings file must be generated by using a sequence of config.ssess_start(), config.set() and config.ssess_commit().

The settings file path on the system is always "/tmp/root/settings.cf2". No other files can be read.

1.7.2.1 Settings file Download

Example config.set() Request to trigger the generation of the cf2 file. Ssess_start() and sess_commit() are not shown here.

```
{
  "id": "req-1",
  "jsonrpc": "2.0",
  "params": [
    "c945b35466f4ffc7c682a875a82d5be9",
    "config",
    "set",
    {
      "cfg_session_id": 1,
      "values": {
        "save_settings_now": "123"
      }
    }
  ]
}
```

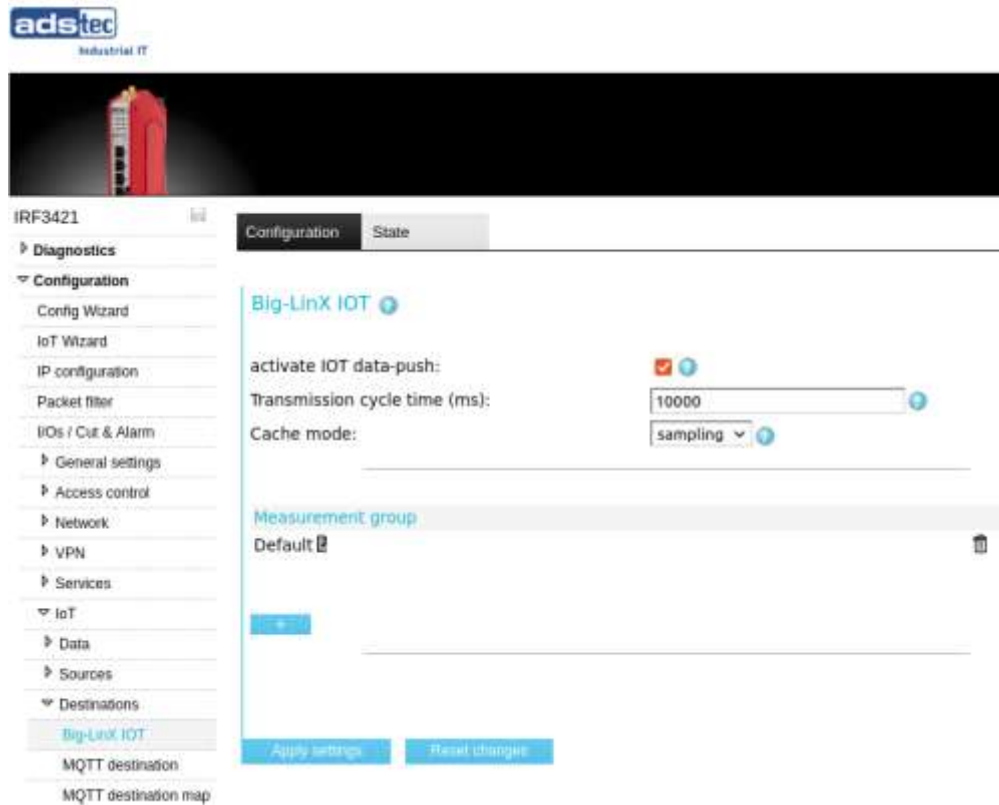
Download the newly generated file:

```
{
  "id": "req-1",
  "jsonrpc": "2.0",
  "method": "call",
  "params": [
    "c945b35466f4ffc7c682a875a82d5be9",
    "file",
    "read",
    {
      "path": "/tmp/root/settings.cf2",
    }
  ]
}
```

The file will remain there until reboot or a new one is generated. To free the used RAM within the system one can delete the file using the file.delete() method using the same path.

1.8 blxpush

The “blxpush” object allows data transfer to Big-LinX IIOT dashboards. Please note that you have to configure a Big-LinX connection on your device first. Afterwards the Big-LinX IOT feature has to be enabled like shown in the following screen shot:



The object on the rpc bus looks like this:

```
'blxpush'  
  "push":{"measurement":"String","tags":"Table","values":"Array"}  
  "status":{}}
```

1.8.1 blxpush.status()

The status method will print the current connection state and the amount of bytes in the buffer. The request will look like this:

```
{
  "id": "1",
  "jsonrpc": "2.0",
  "method": "call",
  "params": [
    "c945b35466f4ffc7c682a875a82d5be9",
    "blxpush",
    "status",
    {}
  ]
}
```

A valid connection with empty buffer should look like this:

```
{
  "jsonrpc": "2.0",
  "id": "1",
  "result": [
    0,
    {
      "status": {
        "token-handle": "available",
        "token": "available",
        "push-handle": "available",
        "queue-content": "0 byte"
      }
    }
  ]
}
```

1.8.2 blxpush.push

The following JSON block shows an example push of acquired data.

```
{
  "id": "1",
  "jsonrpc": "2.0",
  "method": "call",
  "params": [
    "c945b35466f4ffc7c682a875a82d5be9",
    "blxpush",
    "push",
    {
      "measurement": "iiot.test1",
      "tags": {
        "instance": 0
      },
      "values": [{
        "ts": "2022-09-23T13:32:34.119Z",
        "state": 4,
        "p": -2147483648
      }]
    }
  ]
}
```

The response of the system will show whether the data has been accepted or not.

The fields within the example JSON data object have the following meanings:

- measurement: The measurement name inside of the Big-LinX database.
- tags: additional tags to be associated with this data. As an array.
- values: array of integer and float key value pairs. "ts" is a fixed key and the database is expecting a full timestamp value as shown above with 1ms resolution.