Open System Management Protocol

Protocol Version 1.0
Document Version: 2021.06.15
Author: DI Meinrad Recheis, Eqqon GmbH

Table of contents

Introduction

Open System Management Protocol is an open protocol developed by Eqqon in an effort to establish an open protocol that will allow better interoperability between products in (but not necessary limited to) the Public Address domain. The protocol is text-based and uses the open standard JSON for data exchange over a secure WebSocket transport.

Legal Notice

Protocol Design

The protocol is designed to be simple, secure, consistent and well-defined. All communication is encrypted via SSL/TLS (although you can opt out of encryption if both parties agree), the WebSocket connection is regularly monitored by pings preventing half open or dead connections. A continually incremented message sequence number guarantees that either side can always consistently assign received responses to their respective sent commands. This allows commands to be asynchronous, meaning that the reply can be sent immediately or after a very long time while streamed progress messages send feedback about the long running operation and keep the client from timing out while waiting for the operation to complete. A long-running command can be cancelled by the client if necessary. The protocol allows multiple commands to be active at the same time. Server-to-client events ensure immediate state change updates without the need for polling which saves network bandwidth. Last but not least, the protocol is designed to be extensible for the use in all kinds of domains and applications by allowing to extend the standard instruction set by domain specific instruction sets.

Reference Implementation

Eqqon provides an open source reference implementation in C# that can be used for integrating the protocol into third party products.

Domain Specific Instruction Sets

The Open System Management Protocol can be extended with domain specific instruction sets which allow every product which employs OSMP to add or override its own custom commands and instructions. Anyone may create own instruction sets as long as they don’t violate the general protocol specification.

Protocol Specification

Message Format

The WebSocket transport protocol supports three message types (ping, text and binary). However, the Open System Management Protocol is restricted to use only text messages which are formatted using the open-standard JSON. Binary messages and ping messages are simply disregarded. It is important to say that due to the flexible nature of the JSON-format the protocol does not care about the order of the message fields. Also, it is possible to add additional fields to a message without generating errors or influencing the message in any way but size.

General Message Structure

All messages must contain the following data fields except those marked as optional by square brackets [ ]:

Field name	Description
nr	Message sequence number, incremented for every new command. Note: responses will hold the Nr of the command they respond to.
type	One of cmd, response, event, stream
id	Identifier/name of the command, event or stream message.
[data]	The message data (i.e. command parameters) which depend entirely on the type and id of message. Can be omitted if a command has no parameters.

All response messages contain additional data fields:

Field name	Description
cmd-nr	Message sequence number of the command this response is reacting to.
status	Command status. Can be “OK” or “ERROR”
[result]	Contains a detailed result or error message (often omitted for status OK)

Message Types

Messages are differentiated by type. The following message types are defined:

Message type	Description
cmd	Type cmd signifies a command or query to which a more or less immediate response by the other side is expected. Commands are usually sent from the client to the server but server-to-client cmds are also possible so that the server can query information from the client if necessary. Multiple commands can run at the same time. Commands can run a very long time if they start a long-running operation or they may even run indefinitely. While they are running they regularly send stream messages with updates for the client. Long running commands can be cancelled by the cancel message.
response	A response message is an answer to a previously received command. The referenced command name as well as the sequence number of the command message are returned back in the response in order allow unambiguous assignment of a received response to a previously sent command. Note that the sequence number of a response is always the sequence number of the respective command this response is referring to. The response marks the end of a command life-time.
event	An event is a spontaneous message that is usually sent from the server to the client to notify it of changes or happenings without expectation of an answer. There are two types of event messages. Single events and multiple events. If many events are to be sent at the same time they will be packed together in a multi-event message.
cancel	A cancel message is sent by the waiting party (usually the client) to cancel a long running command. The cancelled cmd will acknowledge with a response.
stream	Sent by the receiver of a cmd (usually the server) to send continuous results of a long running operation.

Initiation of a Session

The client initiates a session by simply connecting to the server's secure WebSocket via the address scheme wss://<ip>:<port>/osmp/v1 (i.e. wss://192.168.0.100:443/osmp/v1). The port number can be either the normal SSL port or any other port. In this example case the port is 443. The server responds by sending the *session-initiated* event. In its data, the event contains the protocol name and version.

This is an example of a session-initiated event:

{"type":"event","id":"session-initiated","nr":1,"data":{"protocol":"Open System
Management Protocol","version":1, "public-key":
"<RSAKeyValue>...snipped...</RSAKeyValue>"}}

Command and Response

To send a command create a message of type “cmd” and set the “Id” to the command name. In this example, we use the “help” command from the Standard Instruction Set. The field “Nr” is set to an integer number that is continually incremented with every command the client sends.

The client sends the “help” command …

{"nr":17,"type":"cmd","id":"help"}

to which the server responds with a list of all known commands (truncated for brevity):

{"nr":1, "cmd-nr":17, "type":"response", "id":"help", "data":{
"commands":[
{"command":"apropos", "aliases":["ap"], "instruction_set":"standard",
"description":"Full text search in all available commands for
given keyword."},

{"command":"cookie", "Aliases":[], "instruction_set":"standard",
"description":"Get or set a cookie"},
...
]
},
"status":"OK"}

Note how the server returns a message of type “response” with cmd-nr 17 which refers to the message number of the “help” command the client just sent. This helps the client to differentiate the exact response message to every command it sends without relying on the timing only. Also, the response includes the respective command name in the “Id” field. The “Data” field contains the payload of the response, in this case a list of commands with description and additional info.

Reaction to Invalid Messages

Here is an example how the server reacts to ill-formatted messages. In this case, the client just sent the string “hello?”

{"nr":1,"type":"event","id":"error","data":{"description":"Unable to decode message, wrong formatting? Got this: 'hello?'
****
ArgumentException: \"Unexpected character 'h'. At line 1, column 0\"
---
at … (stacktrace removed for brevity)
****
"}}

Since the server did not get a valid JSON message, it could could not decode it, so instead of a response it sends an event of id “error”.

This is how the server reacts on unknown commands, that is, for instance if the Pizza Instruction Set is not supported:

{"nr":1, "cmd-nr":2, "type":"response", "id":"order-pizza", "data":null, "status":"ERROR", "reason":"Error: command 'order-pizza' not found"}

Command Timeout

A command is expected to return a response within a specific amount of time. The client keeps track of all sent commands and will let those fail which didn’t receive a response or a command stream. Note that asynchronous (long-running) command implementations will not timeout as long as they are active on the server because the server sends a session-status event that reports all active commands to the client.

Command Streams

Usually the client sends a command to the server and gets an immediate response in return. If the server fails to react in time the command times out and fails. Some operations, however, can not be concluded immediately and would time out before the finish processing. Also the client wishes feedback while it waits for the command to finish.

This is where command streams come in. While a command is active, it can send stream messages, which let the client know what is happening, provide regular progress updates or return the requested data piece by piece as it becomes available.

The server supports multiple commands to run at the same time and (like the client) keeps track of all active commands and sends regular session-status events to keep active commands from timing out in the client. When the command finally finishes, it completes by sending their response. Some commands are designed to never finish and thus must be cancelled by the client to conclude. For a detailed example of such a command check out the log-stream command documentation.

The client can distinguish multiple streams by looking at the cmd-nr field which contains the number of the originating command which is still waiting for completion in the client’s waiting commands structure.

Here is an example of two stream messages sent by a log-stream command:

{"type":"stream","nr":5,"cmd-nr":2,"id":"log-stream","data":{"text":"Streaming logger(s) 'Director' >= level DEBUG"}}

{"type":"stream","nr":6,"cmd-nr":2,"id":"log-stream","data":{"text":"Cancel cmd 2 to stop this stream."}}

As you see, both stream messages provide a textual feedback about the command and both contain its cmd-nr 2.

Typical Stream Messages

There are different kinds of streams which differ from each other only in the fields of the data object. The most common streams are:

Stream type	Description
Text fragments	A text stream sends pieces of text. Example data: {"text":"Cancel cmd 2 to stop this stream."}}
Text lines	Similar to a text stream the line stream message transports text. The only difference is that it suggests that a line-break is to be made after this line of text. Example data: {"line":"Cancel cmd 2 to stop this stream."}}
Structured data	A data stream transports structured data and the structure is entirely dependent on the command that sends it. Example data: {"date-time":"2017-08-31T12:14:30.0647371+02:00","logger":"Director","level":"DEBUG","text":"Test log message","thread":10}

Cancelling Commands

Commands that are running longer than the command timeout can be prematurely cancelled by sending the cancel message. Cancellation is also important to end commands which run indefinitely by definition (like *log-stream*).

Here is an example demonstrating the cancellation of a never-ending command:

{"type":"cmd", "nr":1, "id":"log-stream", "data":{"logger":"Director","level":"DEBUG"}}

The command is active now. For a detailed description what it does and what messages are sent while it is active please check out the log-stream command documentation. Now to cancel this command, we have to send the command nr=1 in the cmds array.

{"type":"cancel","nr":2,"id":null,"data":{"cmds":[1]}}

This will end the log-stream command after which it sends its response.

{"type":"response","id":"log-stream","nr":108,"cmd-nr":1,"status":"OK","result":"Log stream cancelled."}

See the documentation of *active-cmds* for a way to query the currently active commands.

Events

Single event messages look like ordinary messages with the id-field being the event name. Multi-event messages have id=null and instead contain an array of events in their data field.

Here is an example of a single event message:

{"type":"event","id":"session-initiated","nr":1,"data":{"protocol":"Open System Management Protocol","version":1}}

… vs an example of a multi-event message:

{"type":"event","id":null,"nr":1,"data":{"events": [
{"event":"session-initiated", “data”:{
"protocol":"Open System Management Protocol","version":1
}}
… there can be many more events in this array
]}

Server to Client Commands

In some situations the server (i.e. while executing a command) needs to ask the client for additional information which the client can not provide ahead of time. A good example of this is the public-key-authentication method. It uses the server command sign-request, which sends a random token to the client to sign with its private key, in order to prove that it is in the possession of the private key.

Server to client commands work exactly like client to server commands. The only difference is, that the client’s immediate response to server commands is expected or else the command fails due to timeout.

Server commands can be thought of as specialized tools for allowing interaction of a normal client command with the client while the command is being executed. They allow client commands a certain degree of interactivity which is needed in some cases. Examples are authentication, file upload and download, user confirmation or user choice.

To avoid infinite command loops a client's server command implementation must not send any commands back to the server.

TODO: Document all server commands

Authentication

There are currently two methods of authentication built into the Standard Instruction Set and implemented by the reference implementation: Password Authentication and Public Key Authentication.
Authentication is optional, so the server will only refuse to execute commands when it is configured to require authentication.

Since the transport channel of the protocol is already encrypted via SSL/TLS all authentication methods are already using a sufficiently secure communication channel, given that the server's SSL certificate is valid. But even if the SSL encryption were to be breached due to certificate problems, authentication is still very secure due to additional RSA encryption of secret information. All secrets are encrypted in combination with a random chunk of information that constantly varies, so that a listener can not use any encrypted secrets they obtained without being detected. This means that both password and public key authentication are extremely secure, especially if RSA keys of at least 1024 bits are used.

Password Authentication

Password authentication is the simplest form of restricting access to the server by logging in with user-password combinations. It is also the easiest to manage if multiple users are to be supported. The server is configured with username:password pairs. It will decline all login requests from unknown users or non-matching passwords.
The password is transferred only in encrypted form. Measures are taken that the password is always encrypted together with a different random token which makes it impossible for an eavesdropper to reuse an encrypted password. As long as the client keeps the passwords secret and access to the server and its private key is sufficiently protected this authentication method is highly secure.

Here is an exemplar communication log between client and server that shows the entire password authorization process. First we see a command to fail due to not having authenticated:

CLIENT: {"type":"cmd","nr":1,"id":"echo","data":null}
SERVER: {"type":"response", "id":"echo", "nr":1, "cmd-nr":1, "status":"ERROR",

"result":"Authentication required!"}

Now client starts password authentication. Note how the password is never transmitted as plaintext:

CLIENT: {"type":"cmd","nr":2,"id":"login","data":{"username":"Jon","method":"password"}}
SERVER: {"type":"cmd","nr":2,"id":"password-request","data":{"token":
"1da99375-0640-4733-8c09-b9ce625a6cb7"}}
CLIENT: {"type":"response","id":"password-request","nr":3,"cmd-nr":2,"status":"OK","data":{
"encrypted-password":"VYcfGyJ3e...snipped...wfJt670zCY="}}
SERVER: {"type":"response","id":"login","nr":3,"cmd-nr":2,"status":"OK","result":"Login
successful!"}

The client is logged in now so commands are working now:

CLIENT: {"type":"cmd","nr":8,"id":"echo","data":null}
SERVER: {"type":"response","id":"echo","nr":8,"cmd-nr":8,"status":"OK"}

Public Key Authentication

Public key authentication utilizes the asymmetric cryptographic algorithm RSA for login verification. Instead of passwords, the server is configured with username:public key pairs and it will decline all login requests from unknown users or not matching public keys. The client sends a login request to the server containing only the username and the client's public key. The server then creates a new random token and sends it back to the client requesting it to sign it using its private key. By doing so and sending back the signature the client proves that it is in possession of the private key for the public key it is logging in with. As long as the client keeps its private key a secret and access to the server is sufficiently protected this authentication method is highly secure.

Here is an exemplar communication log between client and server that shows the entire public key authorization process.

CLIENT: {"type":"cmd","nr":2,"id":"login","data":{"username":"Jon","method":"public-key",
"public-key":"<RSAKeyValue>...snipped...</RSAKeyValue>"}}
SERVER: {"type":"cmd","nr":2,"id":"sign-request","data":{"token":
"2b7ba4ea-6a2b-4a0e-a166-1fe86fd2d989"}}
CLIENT: {"type":"response","id":"sign-request","nr":3,"cmd-nr":2,"status":"OK","data":{
"signature":"jGaec...snipped...8Q9IUY="}}
SERVER: {"type":"response","id":"login","nr":3,"cmd-nr":2,"status":"OK","result":"Login
successful!"}

Standard Instruction Set

The Standard Instruction Set should be supported by every product. It provides a basic infrastructure for interactive help and command search as well as other useful tools like file transfer.

Basic Commands

Command	Description
help	List available commands or display extended command help
apropos	Full text search in all available commands for given keyword.
echo	Returns the given token
time	Returns the current server time
wait	Wait indefinitely (until cancelled) or the given amount of seconds
event-list	List of all events which can be subscribed
event-subscribe	Subscribe a given event or all events
event-unsubscribe	Unsubscribe a given event or all events
active-cmds	Returns list of active commands (excluding own)
login	Request authentication with either username:password or username:public-key
logout	Logout the currently authenticated user

help - List available commands or display extended command help

Without parameters, help will list all available commands. With the optional parameter cmd set it will return detailed help on the given command, including parameter description and examples.

Parameter	Type	Description
cmd	String	The command to return extended help for

Example of a help command with parameter cmd=help

```
{ "type": "cmd", "nr": 1, "id": "help", "data": {"cmd":"help" }}
```

which returns detailed info about the help command’s behavior, its parameters and its return values:

```
{
  "cmd-nr": 1,
  "status": "OK",
  "result": null,
  "type": "response",
  "nr": 1,
  "id": "help",
  "data": {
        "command": "help",
        "aliases": [ "?" ],
        "version": 0,
        "instruction-set": "standard",
        "description": "List available commands or display extended command help.",
        "long-description": "Without parameters, help will list all available commands.
\r\nWith parameter 'cmd' set it will return detailed help on the given command,
including parameter description and return values.",
        "mandatory-params": [],
        "optional-params": [
        {"name": "cmd",
        "description": "The command to print extended help about."}],
        "return-values": [
        {"name": "commands",
        "description": "List of available commands (returned only if parameter cmd is
omitted)"},
        {"name": "command",
         "description": "Name of the optionally specified command"},
        {"name": "aliases",
        "description": "List of aliases for the command"},
        {"name": "version",
        "description": "Command version number"},
        {"name": "instruction-set",
        "description": "Instruction set is the name of the domain specific command
group"},
        {"name": "description",
        "description": "Short description of the command"},
        {"name": "long-description",
        "description": "Longer more detailed description of the command"},
        {"name": "mandatory-params",
        "description": "List of mandatory parameters (must be specified)"},
        {"name": "optional-params",
        "description": "List of optional parameters (can be omitted)"},
        {"name": "return-values",
```

```
            "description": "List of return values"}
            ]
  }
}
```

apropos - Full text search in all available commands for given keyword

The apropos command can help finding all commands that contain a specific keyword in their command descriptions, longtext, parameter or return value documentation.

Parameter	Type	Description
term	String	The search filter (regex ignoring case)

Example of the apropos command with parameter term=announcement

```
{ "type": "cmd", "nr": 4, "id": "apropos", "data": {"term":"announcement" }}
```

Given, that instruction set “vd1” is supported by the server, it returns the following commands containing that term in their documentation:

```
{
  "cmd-nr": 4,
  "status": "OK",
  "result": null,
  "type": "response",
  "nr": 4,
  "id": "apropos",
  "data": {
        "commands": [
              {"cmd": "vcall-create",
              "aliases": ["vcc" ],
              "description": "Create a new or change an existing announcement.",
              "instruction-set": "vd1"},

              … list cut off for sake of brevity.
        ]
  }
}
```

echo - Return the given token

The echo command is mainly for testing. It does nothing other than returning the token if one was given.

Parameter	Type	Description
[token]	Any	A piece of information to be echoed

Example of the echo command:

```
{ "type": "cmd", "nr": 10, "id": "echo", "data": {"token":"Hello World!" }}
```

Response:

```
{ "type": "response", "cmd-nr": 10, "id": "echo", "data": {"token":"Hello World!" }}
```

time - Return the server time

The time command simply returns the current server time (which might be very different from the client time). The command has no parameters.

Example of the time command:

```
{ "type": "cmd", "nr": 10, "id": "time"}
```

Response:

```
{ "type": "response", "cmd-nr": 10, "id": "time",
      "data": {"date-time":"2017-09-05T09:35:36.2418162+02:00" }}
```

wait - Wait indefinitely (until cancelled) or the given amount of seconds

The wait command is mainly used for testing. It simply waits indefinitely until cancelled. If you specify the optional parameter seconds it waits so many seconds and returns then.

Example of the wait command:

```
{ "type": "cmd", "nr": 10, "id": "wait"}
```

As long as it is not cancelled, there is no response. After cancellation a standard response is sent:

```
{ "type": "response", "cmd-nr": 10, "id": "wait", "status": "OK", "result": "Cancelled"  }
```

event-list - List all events including subscription status

The event-list command returns a list of all events which can be subscribed by the client in order to receive them as well as their subscription status.

Example of the event-list command

```
{ "type": "cmd", "nr": 9, "id": "event-list"}
```

The response contains a list of all events including subscription status:

```
{
  "cmd-nr": 9,
  "status": "OK",
  "result": null,
  "type": "response",
  "nr": 9,
  "id": "event-list",
  "data": {
        "now": "2017-08-29T12:51:40.5301082+02:00",
        "events": [
        {
              "event-name": "vcall-status-changed",
              "instruction-set": "vd1",
              "description": "Sent with every call status change",
              "subscribed-until": "2027-05-29T12:33:50.7482151+02:00"
        }
        … list cut off for brevity
        ]
  }
}
```

In this case, the event vcall-status-changed has been subscribed for ten years, which is the default behavior if no subscription timeout is specified in event-subscribe. Note: the field “now” returns the current server time for your info.

event-subscribe - Subscribe a given event or all events

The event-subscribe command will subscribe the client for server-to-client events. This is a necessary precondition for receiving event updates. A subscription lasts only as long as the client server connection lasts. The subscription can be limited in time. The default subscription period is 10 years, so make sure you renew your event subscriptions before that. Note that a subscription automatically ends when the session ends.

To retrieve all event subscriptions and timeouts use the event-list command.

Parameter	Type	Description

event	String	The event to subscribe (* for all events)
[timeout]	TimeSpan	The optional subscription time span

Example of the event-subscribe command to subscribe all events

```
{ "type": "cmd", "nr": 10, "id": "event-subscribe", "data": {"event":"*" }}
```

The server responds with a simple message stating the number of subscribed events.

```
{
  "cmd-nr": 10,
  "status": "OK",
  "result": "Subscribed 2 events.",
  "type": "response",
  "nr": 11,
  "id": "event-subscribe",
}
```

Depending on the kind of event, immediately after subscribing, an event may be sent to notify about the current status of the object or value to be observed by this specific event. In this case the call status for an existing call is immediately transmitted as event:

```
{"type":"event", "nr":10, "data": {"events": [{"event":"vcall-status-changed", "data":
{"date-time":"2017-08-29T14:41:14.9682915+02:00", "call-id":"testcall1",
"status":"Disconnected", "last-started":"0001-01-01T00:00:00"}}]}}
```

See chapter Message Types for detailed description of event messages.

event-unsubscribe - Unsubscribe a given event or all events

The event-unsubscribe command will unsubscribe the client for server-to-client events. To retrieve all event subscriptions and timeouts use the event-list command.

Parameter	Type	Description
event	String	The event to subscribe (* for all events)

Example of the event-subscribe command to subscribe all events

```
{ "type": "cmd", "nr": 10, "id": "event-unsubscribe", "data": {"event":"*" }}
```

The server responds with a simple message stating the number of subscribed events.

```
{
```

```
  "cmd-nr": 10,
  "status": "OK",
  "result": "Unsubscribed 2 events.",
  "type": "response",
  "nr": 11,
  "id": "event-unsubscribe",
}
```

active-cmds - Returns list of active commands

The active-cmds command returns a list of long-running commands which are currently active. See *log-stream* command as an example of a long-running command and *Cancelling Commands* for further info about cancelling active commands.

Example of the active-cmds command:

```
{ "type": "cmd", "nr": 10, "id": "active-cmds" }
```

The response shows one active command:

```
{ "type": "response", "cmd-nr": 10, "id": "echo",
        "data": {
                "now":"2017-08-29T12:51:40.5301082+02:00",
                "cmds": [
                        {
                                "name": "wait",
                                "cmd-nr": 5,
                                "last-started": "2017-08-29T11:51:40.5301082+02:00"
                        }
                ]}}
```

login - Request authentication

The login command communicates the wish of the client to authenticate itself. It is used to initiate both password and public key authentication.

Parameter	Type	Description
username	String	The user account name
method	String	Authentication method (password | public-key)

See chapter Authentication for examples of the login command in action.

logout - Logout the currently authenticated user

The logout command has no parameters and does exactly what you'd expect it to do. See chapter Authentication for an example.

TODO: document rest of the Standard Set.

Director Instruction Set

Director is a PA Management Software by Eqqon. It implements the Open System Management Protocol and a special instruction set to provide easy integration with other applications and third party products.

IO Control Commands

Director provides system agnostic IO control commands to get or set the state of a list of IO contacts.

Command	Description
io-get	Get the state of given IOs
io-set	Set the state of given IOs

io-get - Get the state of given IOs

The io-get command returns a dictionary of { address:state, ... } where address is a system specific IO address and state may be true, false or null if device was not reachable.

Parameter	Type	Description
addrs	Array	List of system specific IO contact addresses.
[system]	String	System (vd1, ied, itec, ambient, multi, remote ... or auto)

Note: every system has its own address scheme which is documented for each relevant system at the bottom of this document.

Example of an io-get command for VD1, getting the first 12 contacts from DAL 1 of DOM 1 (i.e. all the DCS buttons):

```
{ "type": "cmd", "nr": 1, "id": "io-get", "data": {
  "addrs": [
    "lo.1.io.1.1", "lo.1.io.1.2", "lo.1.io.1.3", "lo.1.io.1.4", "lo.1.io.1.5", "lo.1.io.1.6",
    "lo.1.io.1.7", "lo.1.io.1.8", "lo.1.io.1.9", "lo.1.io.1.10", "lo.1.io.1.11", "lo.1.io.1.12"]
}}
```

and the server response to it (buttons 4, 9 and 12 were pressed on the DCS):

```
{
```

```
  "cmd-nr": 4,
  "status": "OK",
  "result": null,
  "type": "response",
  "nr": 7,
  "id": "io-get",
  "data": {
    "rv": {
      "lo.1.io.1.1": false,
      "lo.1.io.1.2": false,
      "lo.1.io.1.3": false,
      "lo.1.io.1.4": true,
      "lo.1.io.1.5": false,
      "lo.1.io.1.6": false,
      "lo.1.io.1.7": false,
      "lo.1.io.1.8": false,
      "lo.1.io.1.9": true,
      "lo.1.io.1.10": false,
      "lo.1.io.1.11": false,
      "lo.1.io.1.12": true
    }
  }
}
```


io-set - Set the state of given IOs

The io-set command allows you to control IOs and returns the resulting state as a dictionary of { address:state, ... } where address is a system specific IO address and state may be true, false or null if device was not reachable.

Parameter	Type	Description
states	Array	List of system specific IO contact addresses.
[system]	String	System (vd1, ied, itec, ambient, multi, remote ... or auto)

Note: every system has its own address scheme which is documented for each relevant system at the bottom of this document.

Example of an io-set command for VD1, setting the first three of DOM 1’s eight contacts:


```
{ "type": "cmd", "nr": 1, "id": "io-set", "data": {
  "states": { "lo.1.io.1": true, "lo.1.io.2": false, "lo.1.io.3": true }
}}
```


and the server response to it (confirming contact 1 and 3 to be on and 2 off):

```
{
  "cmd-nr": 5,
  "status": "OK",
  "result": null,
  "type": "response",
  "nr": 8,
  "id": "io-set",
  "data": {
    "rv": {
      "lo.1.io.1": true,
      "lo.1.io.2": false,
      "lo.1.io.3": true
    }
  }
}
```


Logging Management Commands

Director has extensive logging capabilities and a very flexible high performance logging system. Via the logging management commands logging information with varying degree of detail can be tunnelled to third party clients who are interested. This is especially handy in troubleshooting situations but logging can also be used to get in depth stati or progress updates of important operations.

Command	Description
log-list	List all loggers with level and statistics
log-level	Set level of a logger. If level is not given, return current level
log-stream	Stream the given logger at given level with optional filter applied

log-list - List all loggers with level and statistics

The log-list command returns all known loggers that currently exist in Director.

Parameter	Type	Description
filter	String	Filter the list by logger name

Example of a log-list command:

```
{ "type": "cmd", "nr": 1, "id": "log-list", "data": null }
```

and the server response to it:

```
{
  "status": "OK",
  "type": "response",
  "nr": 1,
  "cmd-nr": 1,
  "id": "log-list",
  "data": {
        "loggers": [
              { "logger": "Director", "level": "ERROR", "count": 0 },
              … list truncated for brevity ...
        ]
  }}
```

log-level - Set level of a logger. If level is not given, return current level

The log-level command is used to query the level of a Director logger or change it.

Parameter	Type	Description
logger	String	Name of the logger (or '*' for all)
[level]	String	Set the logger to the given level (OFF, DEBUG, INFO, WARN, ERROR, FATAL) or DEFAULT to change back to default level.

Example of a log-level command:

```
{ "type": "cmd", "nr": 1, "id": "log-level", "data": {"logger":"Director", "level":"DEBUG"}}
```

and the server response to it:

```
{ "cmd-nr": 1,
  "status": "OK",
  "result": null,
  "type": "response",
  "nr": 1,
  "id": "log-level",
  "data": { "level": "DEBUG" }
}
```

log-stream - Stream the given logger at given level

The log-stream command is used to forward the log messages of a specific Director logger from the server to the client until the command is cancelled using a cancel message. This means, the command can run a very long time in the background while other commands are executed. The protocol is designed to allow multiple commands to be executed at the same time.

Parameter	Type	Description
logger	String	Name of the logger(s) that should be subscribed (comma delimited)
[filter]	String	Filter expression (Regex, multiline, ignores case)
[level]	String	Logger level (default=DEBUG)

Example of a log-stream command:

```
{ "type": "cmd", "nr": 1, "id": "log-stream", "data": {"logger":"Director",
"level":"DEBUG"}}
```

Note: the server does not send a response message right away. Instead it sends a stream message telling the client that the logger stream is subscribed:

```
{"type":"stream", "nr":21, "cmd-nr":1, "id":"log-stream", "data":{"text":"Streaming
logger(s) 'Director' >= level DEBUG"}}
```

and

```
{"type":"stream", "nr":22, "cmd-nr":1, "id":"log-stream", "data":{"text":"Cancel cmd 1 to
stop this stream."}}
```

Also to keep active commands from timing out in the client the server regularly sends a session-status event to inform about all currently active commands. The client uses that information to update the last-receive time stamps of all currently active commands to keep them from timing out while they are active on the server. Here is an example of the session-status event:

```
{"type":"event","nr":23, "id":"session-status","data":{"active-cmds":[1]}}
```

Whenever a log-message is generated the log-stream command streams them to the client like this:

```
{"type":"stream","nr":24,"cmd-nr":1,"id":"log-stream","data":{"date-time":"2017-08-
31T12:14:30.0647371+02:00","logger":"Director","level":"DEBUG","text":"Test log
message","thread":10}}
```

The data fields in the log stream are as follows:

Field	Type	Description
logger	String	Logger name of this log message
date-time	DateTime	The timestamp of this message
level	String	Logger level of this message. Can be DEBUG, INFO, WARN, ERROR or FATAL.
text	String	The log message
thread	Integer	The thread number of the context this log message was generated on

The log-stream cmd streams log messages until it gets cancelled by the client. That means it waits for a cancel message. To stop the stream send a cancel message including the original log-stream command's nr in the cmds array:

```
{ "type": "cancel", "nr": 2, "data": {"cmds":[1]}}
```

or

```
{ "type": "cancel", "nr": 2, "data": {"cmds":"*"}}
```

to cancel all active commands. Now the log-stream command gets cancelled and sends its response message:

```
{"type":"response", "id":"log-stream", "nr":104, "cmd-nr":1, "status":"OK", "result":"Log stream cancelled."}
```

TODO: document rest of the director set.

VD1 Instruction Set

Variodyn D1 is a public address system by Honeywell (both names are trademarks of their owner). Director which is a PA Management Software by Eqqon implements the Open System Management Protocol and provides a VD1 instruction set for management, maintenance and control of Variodyn installations.

Announcement Management Commands

The announcement management commands allow to create and manipulate announcements. They are the core commands needed for paging functionality.
Note: The term *call* is used synonymously for *announcement* throughout Director for brevity and because it is easier to type.

Command	Description
vcall-create	Create a new or change an existing announcement.
vcall-play **vcall-start**	Start an existing announcement
vcall-stop	Stop an existing announcement
vcall-list	List all announcements and show their status.
vcall-status	Query the status of an announcement.
vcall-delete	Delete an existing announcement.

vcall-create - Create a new or change an existing announcement

In order to start an announcement you need to create a call-object and assign a unique id to it which is used by subsequent commands to manipulate it. You can either use a human readable name for the call (like “Evacuation”) or just use a new GUID for every new call.
Note: if you indefinitely create new calls on demand you should keep track of them and use vcall-delete to free up their resources in order to avoid a memory leak.

Parameter	Type	Description
call-id	String	Unique identifier of the announcement to be created (or manipulated). The id is chosen by the client.
[sources]	Array of Strings	List of audio sources to play in sequence
[zones]	Array of Strings	List of zones to announce to

[priority]	Integer (1..255)	Announcement priority. Calls with lower priority number overrule calls with lower priority. 20..40 are alarm priorities, 41 is default, 41..255 are normal call priorities
[owner]	String	Informative id or name of the client which created this announcement
[start]	Boolean	Set this to true if the call should be started immediately. Otherwise, the call object will just be created and can be started by a subsequent vcall-play command.

Example of a vcall-create command:

```
{"nr":2, "type":"cmd", "id":"vcall-create", "data":{
        "call-id":"testcall1",
        "sources":["EvacuationText"],
        "zones":["Floor 3", "Floor 4", "Lobby", "Staircase"],
        "priority":42
}}
```

and the server response to it:

```
{"cmd-nr": 2, "nr": 2, "type": "response", "id": "vcall-create",
    "data": {
        "call": {
            "call-id": "testcall1",
            "sources": ["EvacuationText"],
            "zones": ["Floor 3", "Floor 4", "Lobby", "Staircase"],
            "priority": 42,
            "owner": null
        }
    },
    "status": "OK"}
```

The vcall-create command can also be used to edit just one data field of an existing call, like for instance, if we wanted to play a gong before the evacuation text:

```
{"nr":3, "type":"cmd", "id":"vcall-create", "data":{
        "call-id":"testcall1",
        "sources":["Gong", "EvacuationText"],
}}
```

The response will now show, that only sources has been changed, all other properties are still as they were:

```
{"cmd-nr": 3, "nr": 3, "type": "response", "id": "vcall-create",
    "data": {
        "call": {
```

```
                        "call-id": "testcall1",
                        "sources": ["Gong", "EvacuationText"],
                        "zones": ["Floor 3", "Floor 4", "Lobby", "Staircase"],
                        "priority": 42,
                        "owner": null
                    }
                },
                "status": "OK"
}
```


vcall-play alias vcall-start - Start an existing announcement

The vcall-play command and its alias vcall-start are used to start an existing call that has previously been created with vcall-create. If the call is already playing, this is a no-op. Note that the announcement will end on its own if the sources are all finite (i.e. pre-recorded texts). If the sources contain an infinite audio source (i.e. an audio input or a generated signal) the announcement will play indefinitely until it is stopped with vcall-stop.

Parameter	Type	Description
call-id	String	Unique identifier of the announcement to be started

Example of a vcall-play command:


```
{"nr":4, "type":"cmd", "id":"vcall-play", "data":{ "call-id":"testcall1", } }
```


The server responds with status OK.


```
{"cmd-nr": 4, "nr":4, "type":"response", "id":"vcall-play", "data":{}, "status":"OK"}
```


Or, if the call does not exist:


```
{"cmd-nr": 4, "nr":5, "type":"response", "id":"vcall-play", "data":{}, "status":"ERROR",
"reason":"No such call: testcall2"}
```


vcall-stop - Stop an existing announcement

The vcall-stop command is used to stop an existing call that has previously been started with vcall-play or vcall-start. If the call is not playing this is a no-op.

Parameter	Type	Description
call-id	String	A unique identifier by which the call can be addressed (or

		* for all)

Example of a vcall-stop command:

```
{"nr":6, "type":"cmd", "id":"vcall-stop", "data":{ "call-id":"testcall1", } }
```

The server responds with status OK.

```
{"cmd-nr": 6, "nr":6, "type":"response", "id":"vcall-stop", "data":{}, "status":"OK"}}
```

vcall-list - List all announcements and show their status

The vcall-list command is used to get a complete or partial list of all announcements. There is an optional filter field against which the call's fields are tested against in order to return only those calls which match the expression with at least one field. The filter expression will be interpreted as a C# regular expression. Omitting the filter will return all calls.

Parameter	Type	Description
[filter]	String	Optional regex filter to match the announcements against

Example of a vcall-list command:

```
{"nr":8, "type":"cmd", "id":"vcall-list", "data":{}}
```

The server returns the list of all known announcements, in this case two calls:

```
{"cmd-nr": 8, "nr":8, "type":"response", "id":"vcall-list",
        "data":{
                "calls":[
                        {"call-id":"testcall1", "sources":["EvacuationText"],
                        "zones":["Floor 3","Floor 4","Lobby","Staircase"], "priority":100,
                        "owner":null, "status":"Disconnected"},
                        {"call-id":"testcall2", "sources":["Gong","EvacuationText"],
                        "zones":["Floor 1","Floor 2","Lobby","Staircase"], "priority":0,
                        "owner":null, "status":"Disconnected"}]
        },
        "status":"OK"}
}
```

The fields *status* and its values is described with the *vcall-status* command.

vcall-status - Query the status of an announcement

The vcall-status command returns the status an existing announcement.

Parameter	Type	Description
call-id	String	A unique identifier by which the call can be addressed.

Example of a vcall-status command:

```
{"nr":9, "type":"cmd", "id":"vcall-status", "data":{"call-id":"testcall1", }}
```

The server returns the call stati of the announcement *testcall1*:

```
{"cmd-nr": 9, "nr":9, "type":"response", "id":"vcall-status",
        "data":{
                "call":{"call-id":"testcall1", "status":"Connected", }
        },
        "status":"OK"
}
```

The possible values for *status* and their meaning are:

status	Description
Disconnected	The call is not active
Connected	The call has been started and is playing to all zones
PartiallyConnected	The call has been started and is playing to some zones
Interrupted	The call is active but currently not able to play either due to higher priority calls overruling it or due to a resource conflict.

vcall-delete - Delete an existing announcement.

The vcall-delete command deletes an existing announcement, stopping it in the process if necessary and frees all resources the call used up. It is important to note, that it is important to do so to free up memory in a scenario where new calls are continually and dynamically created by the consumer of the interface over a long period of time. It is the duty of the creator of a call to keep track and delete them once they aren't required any more to avoid a memory leak.

Parameter	Type	Description
call-id	String	A unique identifier by which the call can be addressed (or * for all).

Example of a vcall-delete command:

```
{"nr":10, "type":"cmd", "id":"vcall-delete", "data":{"call-id":"*", }}
```

The server returns the number of deleted calls in the result field:

```
{"cmd-nr": 10, "nr":10, "type":"response", "id":"vcall-delete",
"result":"2 calls deleted.", "status":"OK"}
```

Device Status Commands

The device status commands allow to query the current status of Variodyn controllers and their sub-devices such as amplifiers, call stations, speaker lines, etc. and to configure the notification filters to reduce the status event load (see vdevice-status-changed event in chapter VD1 Events). The following commands are defined:

Command	Description
vdevice-status	Query the status of one or many devices
vdevice-status-filter	Configure filter settings for event vdevice-status-changed

vdevice-status - Query the status of one or many devices

The vdevice-status command is used to get the status of a specific device, for all known devices or for a partial list of devices of interest.

Field	Type	Description
device-id	String	Controller or device address (* for all)
[ignore-not-installed]	Boolean	Don't return devices with status NotInstalled (applied only if device-id is *).
[include-filter]	String	Regex specifying which devices to include, omit to include all (applied only if device-id is *)
[exclude-filter]	String	Regex specifying which devices to exclude (applied only if device-id is *)

Note that the filter and ignore settings are valid only for the current command execution. To change the filter settings that are applied to vdevice-status event notifications see vdevice-status-filter.

Here is an example of a vdevice-status command:

```
{"type": "cmd", "nr": 5, "id": "vdevice-status", "data": {
        "device-id": "*",
        "ignore-not-installed": "True",
        "exclude-filter": "\\.(pr|lr)\\.",
        "include-filter": "lo\\.64"
}}
```


The response gives a partial list of stati that includes only sub-devices of DOM 64 disregarding stati of line relays and pre-amps and of disabled devices. Obviously our DOM 64 has a problem with the amplifiers and only one of the two call stations is working.


```
{"type":"response","id":"vdevice-status","nr":6,"cmd-nr":5,"status":"OK","data":{"stati":[
        {"device-id":"lo.64.sc.0","status":"Error"},
        {"device-id":"lo.64.ds.1.1","status":"Ok"},
        {"device-id":"lo.64.ds.2.1","status":"Error"},
        {"device-id":"lo.64.pa.41.1","status":"Ok"},
        {"device-id":"lo.64.pa.42.1","status":"Ok"},
        {"device-id":"lo.64.pa.43.1","status":"Ok"},
        {"device-id":"lo.64.pa.44.1","status":"Ok"},
        {"device-id":"lo.64.pa.1.1","status":"Ok","sub-status":"AmplificationLow"},
        {"device-id":"lo.64.pa.2.1","status":"Ok","sub-status":"AmplificationLow"},
        {"device-id":"lo.64.pa.3.1","status":"Ok","sub-status":"AmplificationLow"},
        {"device-id":"lo.64.pa.4.1","status":"Ok","sub-status":"AmplificationLow"}
]}}
```


The possible (main) status values are:

status	Description
Online	The system controller (i.e. DOM, SCU) is currently reachable on the network. Online can be overruled by status Error it the controller is online but has errors in its sub-devices
Offline	System controller (i.e. DOM, SCU) is not reachable on the network
Ok	Sub-device (pa, lr, ds, etc) is working and has no errors.
Error	For controllers Error means the device is online but has errors in one or many sub-devices. For sub-devices it means the device has a serious problem. This usually means that PA functionality is affected. The sub-status indicates what is wrong with the sub-device (like MicDefect)
NotInstalled	Will be returned as the status of sub-devices which are disabled by the controller configuration.

These sub-status values specify the reason for the main status Error:

status	Description
MicDefect	Means that the DCS microphone is defect
AmplificationLow	Amplifier warning
AmplificationHigh	Amplifier warning
DistortionHigh	Amplifier warning (total harmonic distortion too high)
NoSetpoint	Line monitoring warning
ImpedanceHigh	Line monitoring error (line broken?)
ImpedanceLow	Line monitoring error (line shorted?)

vdevice-status-filter - Configure device status filter

The vdevice-status-filter command allows you to change the filter settings for the event vdevice-status-changed. This is important because the number of devices and sub-devices in a VD1 system installation can be huge. Using this filter, you can discard devices that are not relevant to your specific application. Per default, the filter is configured to exclude status information about disabled sub-devices (status NotInstalled) which are usually of no importance at all.

Field	Type	Description
[ignore-not-installed]	Boolean	Don't notify about devices with status NotInstalled if True
[include-filter]	String	Regex specifying which devices to include, set null to include all
[exclude-filter]	String	Regex specifying which devices to exclude, set null to exclude none

All command fields are optional which means, that omitting them will not change the settings but only return their values.

Here is an example of a vdevice-status-filter command to query the settings:

```
{"type": "cmd", "nr": 5, "id": "vdevice-status-filter", "data": {}}
```

The response returns the current settings:

```
{"type":"response","id":"vdevice-status-filter","nr":9,"cmd-nr":3,"status":"OK",
"data":{"ignore-not-installed":true,"include-filter":null,"exclude-filter":"\\.(pr|lr)\\."}}
```

VD1 Events

The VD1 instruction set defines the following events:

Command	Description
vcall-status-changed	Informs about announcement status changes
vdevice-status-changed	Notifies about status changes

vcall-status-changed - Informs about announcement status changes

The vcall-status-changed event is sent whenever a call created by vcall-create changes its status.

Field	Type	Description
call-id	String	Call id
status	String	Call status (Disconnected, Connected, PartiallyConnected or Interrupted).
last-started	DateTime	Time stamp of last start
date-time	DateTime	Time stamp of the change

For more info about the status values see the vcall-status command.

Here is an example of a vcall-status-changed event:

{"type":"event", "nr":10, "data": {"events": [{"event":"vcall-status-changed", "data": {"date-time":"2017-08-29T14:41:14.9682915+02:00", "call-id":"testcall1", "status":"Disconnected", "last-started":"0001-01-01T00:00:00"}}]}}

vdevice-status-changed - Informs about status changes

The vdevice-status-changed event is sent whenever a Variodyn device status change is detected by Director. The ‘stati’ field of the response is an array of status objects for every device (that is not excluded by the vdevice-status-filter). Since there are potentially a lot of devices, this event can be filtered. See command vdevice-status-filter for details.

Depending on the number of system controllers, status change detection can take some time because Director is forced to poll all the devices or (in case of controller online status) will hold back for some seconds to verify the status as not to create false alarm due to short-lived communication problems.

Note: since a large Variodyn installation can hold thousands of devices, this change notification is differential. When first subscribing to the event the client will get a complete status of all known and properly configured devices (controllers, amplifiers, call stations, line relays, etc). All subsequent updates are only differential. **In order to avoid inconsistencies it is strongly advised to regularly resubscribe the event** (i.e. hourly) in order to get another complete status snapshot or to regularly query the current status of all devices using command vdevice-status.

Field	Type	Description
stati	Array	List of device status objects
date-time	DateTime	Time stamp of the change

The device status objects are defined as follows:

Field	Type	Description
device-id	String	Address of the controller or sub-device.
status	String	Main status of the controller or device. This is one of Online, Offline, Error, Ok, NotInstalled
[sub-status]	String	Optional sub-status that explains the reason for main status Error or gives additional warnings even when device is status Ok. The sub-status can be one of AmplificationLow, AmplificationHigh, ValueLow, ValueHigh, NoSetpoint, NotInstalled, MicDefect, ImpedanceHigh, ImpedanceLow, DistortionHigh

For more info about the meaning of the status and sub-status values see the vdevice-status command.

Here is an example of a vdevice-status-changed event:

{"type":"event", "nr":2, "id":"vdevice-status-changed", "data":
{"date-time":"2017-09-21T14:11:11.8058743+02:00", "stati":[{"device-id":"lo.200.sc.0","status":"Online"}, {"device-id":"lo.201.sc.0","status":"Offline"}, {"device-id":"lo.64.lr.4.3","status":"Ok", "sub-status": "NoSetpoint"}, {"device-id":"lo.64.ds.1","status":"Error", "sub-status": "MicDefect"},]}}

The device addresses follow the typical Variodyn address scheme employed throughout Director. Check out chapter Director-VD1 Address Scheme for more info.

Director-VD1 Address Scheme

All devices in Variodyn can be addressed by the following address scheme ...

lo.<sysno>.<devtype>.<nr>[[.0].<subnr>]

- **sysno** is the controller number (i.e. DOM Nr),
- **devtype** is one of the following (only most important types are listed)

devtype	Description
sc	System controller (DOM, SCU, Director, etc)
ho	Host, a client talking to a system controller (Director)
pa	Amplifier
pr	Pre-Amplifier
lr	Line relay
ds	Call station (i.e. DCS or UIM)
ao	Audio output
ai	Audio input

- **nr** is the device number.
- **.0** is only relevant for audio titles and recording slots (ai and ao)
- **subnr** Some devices like pa, lr have sub-numbering, some don't

Not all devices need a subnr. For easier orientation, just check out the tables of most important device addresses down below to know which devices are to be addressed how.

Devices

This table shows controller and sub-device addresses (assuming a controller with nr 200 for the sub-device addresses). These are typically used for device status updates in Director:

address (range)	Description
lo.1.sc.0 - lo.250.sc.0	All possible system controller addresses, with system nr 1 through 250 (PA Protocol V2)
lo.200.pa.1.1 - lo.200.pa.4.1	Amplifier channels
lo.200.lr.1.1 - lo.200.lr.1.6	Line relays on channel 1
lo.200.lr.2.1 - lo.200.lr.2.6	Line relays on channel 2
lo.200.lr.3.1 -	Line relays on channel 3

lo.200.lr.3.6	
lo.200.lr.4.1 - lo.200.lr.4.6	Line relays on channel 4
lo.200.ds.1 - lo.200.ds.4	DCS or UIM on DAL 1 - 4
lo.200.pa.41.1 - lo.200.pa.44.1 -	Speaker (Output 1) amp on DAL 1 - 4
lo.200.pa.41.2 - lo.200.pa.44.2 -	Output 2 amp on DAL 1 - 4
lo.200.pr.41.1 - lo.200.pr.44.1 -	Mic (Input 1) pre-amp on DAL 1 - 4
lo.200.pr.41.2 - lo.200.pr.44.2 -	Input 2 pre-amp on DAL 1 - 4
lo.200.pr.1 - lo.200.pr.4	DOM Input 1 - 4 pre-amplifiers

Outputs

Assuming a controller with nr 200, this table shows the most important outputs. These are used to define zones in Director:

address (range)	Description
lo.200.ao.1 - lo.200.ao.6	Channel 1 lines
lo.200.ao.9 - lo.200.ao.14	Channel 2 lines
lo.200.ao.17 - lo.200.ao.22	Channel 3 lines
lo.200.ao.25 - lo.200.ao.30	Channel 4 lines
lo.200.ao.51 - lo.200.ao.58	DOM contacts
lo.200.ao.41	Speaker on DAL 1
lo.200.ao.42	Output 2 on DAL 1
lo.200.ao.43	Speaker on DAL 2
lo.200.ao.44	Output 2 on DAL 2

lo.200.ao.45	Speaker on DAL 3
lo.200.ao.46	Output 2 on DAL 3
lo.200.ao.47	Speaker on DAL 4
lo.200.ao.48	Output 2 on DAL 4

Inputs

Assuming a controller with nr 200, this table shows the most important inputs. These are used to define audio sources in Director:

address (range)	Description
lo.200.ai.1 - lo.200.ai.4	DOM microphone inputs
lo.200.31.0.1 - lo.200.31.0.27	DOM signal slots (chimes, sinus, alarms, etc). Here are some very important examples: lo.200.ai.31.0.1 = Chime lo.200.ai.31.0.2 = Chime 2 lo.200.ai.31.0.3 = Chime 3 lo.200.ai.31.0.17 = Pink Noise lo.200.ai.31.0.26 = Sinus 1kHz
lo.200.ai.41	Mic on DAL 1
lo.200.ai.42	Input 2 on DAL 1
lo.200.ai.43	Mic on DAL 2
lo.200.ai.44	Input 2 on DAL 2
lo.200.ai.45	Mic on DAL 3
lo.200.ai.46	Input 2 on DAL 3
lo.200.ai.47	Mic on DAL 4
lo.200.ai.48	Input 2 on DAL 4

IOs

Assuming a controller with nr 200, this table shows the possible IO contact addresses.

address (range)	Description
lo.200.io.1 - lo.200.io.8	DOM internal contacts
lo.200.io.1.1 - lo.200.io.1.48	DCS/UIM contacts on DAL 1
lo.200.io.2.1 - lo.200.io.2.48	DCS/UIM contacts on DAL 2
lo.200.io.4.1 - lo.200.io.4.48	DCS/UIM contacts on DAL 3
lo.200.io.4.1 - lo.200.io.4.48	DCS/UIM contacts on DAL 4