AS1620: XML-RPC interface

1. Document revisions

Rev	Date	Name	Description
r00	28/06/2017	FG	First version

Contents

2. Introduction

Purpose

This document describes the API (Application Programming Interface) that is available on the VOPAC board. Clients can use this API to manage and control any barrier using on a VOPAC board. This API offers the following features:

- Get and set operational modes;
- Get the technical defects;
- Send commands (open, close);
- Get and set the counters;

This API is implemented as a XML-RPC API and it can be called remotely through an HTTP connection. Clients can be written in a wide variety of programming languages as libraries implementing the XML-RPC protocol are available.

What is RPC?

RPC stands for Remote Procedure Call. As its name indicates it is a mechanism to call a procedure or function available on a remote computer. Remote Procedure Call (RPC) is a much older technology than the Web. Effectively, RPC gives developers a mechanism for defining interfaces that can be called over a network. These interfaces can be as simple as a single function call or as complex as a large API.

What is XML-RPC?

XML-RPC is among the simplest and most fool-proof web service approaches, and makes it easy for computers to call procedures on other computers.

XML-RPC permits programs to make function or procedure calls across a network.

XMLRPC uses the HTTP protocol to pass information from a client computer to a server computer.

XML-RPC uses a small XML vocabulary to describe the nature of requests and responses.

XML-RPC client specify a procedure name and parameters in the XML request, and the server returns either a fault or a response in the XML response.

XML-RPC parameters are a simple list of types and content - structs and arrays are the most complex types available.

XML-RPC has no notion of objects and no mechanism for including information that uses other XML vocabularies.

With XML-RPC and web services, however, the Web becomes a collection of procedural connections where computers exchange information along tightly bound paths.

XML-RPC emerged in early 1998; it was published by UserLand Software and initially implemented in their Frontier product.

Why XML-RPC?

If you need to integrate multiple computing environments, but don't need to share complex data structures directly, you will find that XML-RPC lets you establish communications quickly and easily.

Even if you work within a single environment, you may find that the RPC approach makes it easy to connect programs that have different data models or processing expectations and that it can provide easy access to reusable logic.

XML-RPC is an excellent tool for establishing a wide variety of connections between computers.

XML-RPC offers integrators an opportunity to use a standard vocabulary and approach for exchanging information.

XML-RPC's most obvious field of application is connecting different kinds of environments, allowing Java to talk with Perl to talk with Python to talk with ASP, and so on.

XML-RPC References

XML-RPC is a well-documented protocol. Numerous books have been published on this topic. See for example the following book published by O'REILLY:

> ***PROGRAMMING WEB SERVICES WITH XML-RPC***
> ***AUTHORS: S. ST. LAURENT, J. JOHNSTON & E. DUMBILL***
> ***ISBN 0-596-00119-3.***

Programming of a XML-RPC client can be simplified by using libraries that implements the protocol. XML-RPC libraries exist for a wide range of programming languages including C/C++, Java, PHP, Perl, Python, Ruby, etc.

3. API Reference

Convention

The available XML-RPC functions are documented according to the following convention:

```
Results = function_name(Args)
```

Where:

- `Args` are the function arguments. `Args` is an array of integer or string.
- `Results` are the values returned by the function call. `Results` is an array of integer, string, structure or array.

GetSoftware()

Description: Get the software version and revision

Arguments: NIL

Results:

`Results[0]`	(struct) structure describing the software

Where software struct is:

`Struct['sVersion']`	(string) Software release version (e.g. V01R02)
`Struct['iSvn']`	(int) Software SVN version
`Struct['iHWrev']`	(int) Hardware revision
`Struct['sBoard']`	(string) Board name (AS1620R04)

`Struct['sSN']`	(string) Board serial number
`Struct['sDate']`	(string) Board production date (YYMMDD)

GetProperties()

Description: Get the name assigned to the device, the group that contains the device, and the position of the device in the group. These properties are used by the monitoring panel.

Arguments: NIL

Results:

`Results[0]`	(struct) structure describing the properties

Where:

Properties struct is:

`Struct['sType']`	(string) type of the device, see section 4
`Struct['sDeviceName']`	(string) name of the device
`Struct['sGroupName']`	(string) name of the group that contains the device
`Struct['iPosition']`	(int) position of the device in the group
`Struct['iLanePos']`	(int) Code with position in the lane, see section 4
`Struct['iRole']`	(int) Code with role in the lane, see section 4
`Struct['iPartner']`	(int) position of the partner device in the group (if any)

SetProperties()

Description: Set the given properties

Arguments:

`Args[0]`	(string) name of the device
`Args[1]`	(string) name of the group that contains the device
`Args[2]`	(int) position of the device in the group [1,72]
`Args[3]`	(int) Code with position in the lane, see section 4
`Args[4]`	(int) Code with role in the lane, see section 4
`Args[5]`	(int) position of the partner device in the group [1,72]

Results:

`Results[0]`	(int) 1 in case of success, 0 if it failed

GetOpModeList()

Description: Get the list of all operation modes that are supported by the barrier, for instance “automatic”, “free access”, “blocked open”, “blocked close”.

Arguments: NIL

Results:

`Results[0]`	(int) Number of operation mode (N)
`Results[1][0..N-1]`	(array of struct) list of supported operating modes

Where:

Mode struct is

struct['iValue']	(int) Mode id
struct['sName']	(string) Language code describing the mode, see section 4

GetOpMode()

Description: Get the current operating mode.

Arguments: Nil

Results:

Results[0]	(int) id of the operating mode (see GetOpModeList)

SetOpMode()

Description: Set the given operating mode.

Arguments:

Args[0]	(int) Id of the operating mode (see GetOpModeList)

Results:

Results[0]	(int) 1 in case of success, 0 if it failed

GetStatus()

Description: Get status information.

Arguments: NIL

Results:

Results[0]	(bool) Device in maintenance
Results[1]	(bool) Barrier is open
Results[2]	(bool) Barrier is close
Results[3]	(bool) Arm is moving
Results[4]	(bool) Emergency
Results[5]	(int) Operation mode
Results[6]	(int) Number of technical defects (Nd)
Results[7][0..Nd-1]	(array of struct) list of defects
Results[8]	(int) Number of inputs (Ni)
Results[9][0..Ni-1]	(array) list of inputs values
Results[10]	(int) Number of outputs (No)
Results[11][0..No-1]	(array) list of outputs values

Where

Defect struct is

struct['sType']	(string) Language code describing the defect, see section 4
struct['iIsMajor']	(int) Severity [1=major, 0=minor]

GetInputs()

Description: Get the description and value of each input

Arguments: Nil

Results:

`Results[0]`	(int) Number of inputs (N)
`Results[1][0..N-1]`	(array of struct) description of each input

Where:

Input struct is

struct`['iValue']`	(int) current value
struct`['sType']`	(string) Language code describing the input, see section 4

GetOutputs()

Description: Get the description and value of each output

Arguments: Nil

Results:

`Results[0]`	(int) Number of outputs (N)
`Results[1][0..N-1]`	(array of struct) description of each output

Where:

Output struct is

struct`['iValue']`	(int) current value
struct`['sType']`	(string) Language code describing the input, see section 4
struct`['sAlias']`	(string) Alias (if any) given by user

SetOpen()

Description: Send an opening command.

Arguments: Nil

Results:

`Results[0]`	(int) 1

SetClose()

Description: Send a closing command.

Arguments: Nil

Results:

`Results[0]`	(int) 1

SetEmergency()

Description: Enable or disable the evacuation mode. This is equivalent to temporary set the barrier in “blocked open” mode.

Arguments:

`Args[0]`	(int) 0=disabled, 1=enabled

Results:

`Results[0]`	(int) 1

GetEmergency()

Description: Get the evacuation mode current status.

Arguments: Nil

Results:

`Results[0]`	(int) 0=disabled, 1=enabled

GetCounters()

Description: Get the value of all the counters.

Arguments: Nil

Results:

`Results[0]`	(int) Num of counters (Nc)
`Results[1][0..Nc-1]`	(array of struct) list of counters

Where:

Counter struct is

struct`['sName']`	(string) Language code describing the counter, see section 4
struct`['iPerp']`	(int) Perpetual counter, never reset
struct`['Main']`	(int) Main counter, can be reset

SetCounter()

Description: Set the value of a counter. The value can be 0 (reset counter) or any positive number for the counters that are settable.

Arguments:

`Args[0]`	(int) id of the counter (see GetCounters() and section 4)
`Args[1]`	(int) new value of the counter

Results:

`Results[0]`	(int) 0=error; 1=counter set to new value;

4. Language Codes

List of codes used:

Properties: position in the lane

- 0 = Front Left
- 1 = Back Left
- 2 = Front Right
- 3 = Back Right

Properties: role in the lane

- 0 = Independent
- 1 = Master
- 2 = Slave

Device types

- BL15
- BL229 : Standard BL229
- BL229H : BL229 for highways
- BL40
- BL41
- BL43
- BL44
- BL45
- BL46
- BL47
- BL5X
- BL77

Operation modes:

- AUTO : Automatic mode
- FREE : Free access mode
- BLOP : Locked open mode
- BLCL : Locked closed mode

List of defects:

- E-00 and E-10 : Motorisation
- E-01 and E-11 : Sensor position
- E-02 and E-12 : Installation
- E-03 and E-13 : Blocked movement
- E-04 and E-14 : Temperature
- E-05 and E-15 : Inputs/Outputs
- E-06 and E-16 : Snap Out

Input types:

- IN00 : Disabled

- IN01 : Open
- IN02 : Open B
- IN03 : Close
- IN04 : Open and Close
- IN05 : Authorization Terminal
- IN06 : Authorization Terminal B
- IN09 : Stop
- IN10 : Snap Out Sensor
- IN11 : Local Mode
- IN12 : Lock open
- IN13 : Lock closed
- IN14 : Lock boom
- IN15 : Inhibit
- IN16 : Inhibit B side
- IN17 : Reset Capacity
- IN18 : Passage
- IN19 : Passage B
- IN22 : Validation Loop
- IN23 : Validation Loop B
- IN24 : Open Loop
- IN25 : Open Loop B
- IN26 : Close Loop
- IN27 : Security Loop
- IN28 : Limit Switch Opened
- IN29 : Limit Switch Closed
- IN30 : Power Fail Detection
- IN31 : Dead Bolt Check
- IN32 : Motor Off
- IN33 : Manual Mode
- IN34 : Open (Local)
- IN35 : Close (Local)
- IN36 : Reset Counter 1
- IN37 : Increment Counter 1
- IN38 : Decrement Counter 1
- IN39 : Reset Counter 2
- IN40 : Increment Counter 2
- IN41 : Decrement Counter 2

Output types:

- O00 : Not Used
- O01 : Copy Input
- O02 : Passage Contact
- O03 : Logical Combination
- O04 : End Position
- O05 : Parking Full
- O06 : Open Command
- O07 : Close Command
- O08 : Stop Command

- O09 : Engine Brake
- O10 : Electromagnetic Tip Support
- O11 : Dead Bolt
- O12 : Green Lights
- O13 : Green Lights A
- O14 : Green Lights B
- O15 : Red Lights
- O16 : Red Lights A
- O17 : Red Lights B
- O18 : Boom Lights
- O19 : Active Security
- O20 : Block Mantrap
- O21 : Fraud Detection
- O22 : Counter 1 Threshold
- O23 : Counter 2 Threshold
- O24 : Custom

Counters:

- CN00 : Cycles
- CN04 : Opening commands
- CN05 : Motor emergency stops
- CN06 : Major errors
- CN07 : Boom snapped out
- CN08 : Passages
- CN09 : Passages from opposite direction
- CN10 : Vehicles in lot (settable)
- CN11 : Number of Authorizations (settable)
- CN12 : Counter 1 (settable)
- CN13 : Counter 2 (settable)

5. How-to

How to connect to the XML-RPC server

Physical interface: Ethernet

Protocol: XML/RPC

Port: 8081

Python

The following is written in Python 3 using the standard library xmlrp. It shows how to connect to the VOPAC XML-PRC server and how to use the different calls that are provided:

```
import xmlrpc.client


def run(IPaddress,port=8081):
    URI="http://{0}:{1}".format(IPaddress,port)
    with xmlrpc.client.ServerProxy(URI) as proxy:
        results=proxy.GetStatus()
        print(results)


if __name__ == "__main__":
    IPaddress=input("IP address: ")
    run(IPaddress)    run(IPaddress)
```

C++

The following is written in C++ using the standard library XmlRpc++ (http://xmlrpcpp.sourceforge.net/).

```
#include <iostream>
#include <stdlib.h>
#include "XmlRpc.h"

int main(int argc, char **argv)
{
      if (argc != 2)
      {
            std::cerr << "Usage: "
            std::cerr << << argv[0] << " serverHost\n";
            return -1;
      }
      const char* hostname = argv[1];
      int port = 8081;
      // Create a client and connect to the server at hostname:port
      XmlRpc::XmlRpcClient connection(hostname, port);
      XmlRpc::XmlRpcValue args, result;

      // Send command
      if (connection.execute("GetStatus", args, result))
      {
            std::cout << result << std::endl;
      }
      connection.close();
      return 0;
}
```