ProgramPasser RPC

Remote Manipulation of Wash Program Queue

Author : Jaap Versteegh <j.r.versteegh@orca-st.com>
Created : 2009-11-19
Edited : 2023-05-28
Revision : 80
API version: 9

Table of Contents

1 Introduction

This document contains information on the RPC interface of ProgramPasser. ProgramPasser is an ACE software program that interfaces with various carwash controllers in order to make them execute the wash programs that were sold at the cash register without further intervention from personnel.
ProgramPasser maintains a queue of wash programs. When a wash program is first in the queue it is sent to the carwash controller, either immediately or when a car is detected. After the controller has accepted the program for execution, the wash program is removed from the queue.
Through the RPC interface, wash programs can be added to the queue, the queue contents can be retrieved and programs can be deleted and inserted in order to fix the queue when it got out of sync for some reason.
Wash programs can also be added to the queue by scanning a specially formatted barcode, which can e.g. be printed on a customer's receipt.
First ProgramPasser is described in a little more detail. Then the application layer for the RPC is discussed, followed by the RPC API. Examples are provided for clarity. Finally the format for the barcode is described.

2 ProgramPasser

ProgramPasser is a mini carwash controller. It maintains a queue of wash programs, can monitor the gate and pulse signals from the carwash tunnel through digital inputs and track the position of the cars in the tunnel.
It's main purpose however is to act as a proxy between cash register software and actual carwash controllers. It can switch either a digital output associated with a program number or post information through RS232 or TCP to carwash controllers in order to operate them.
It has a communication interface so other software can manipulate the program queue or receive a notification when a car arrives at a certain position in the carwash tunnel. The latter is useful for e.g. displaying messages to customers on displays in or at the end of the carwash tunnel with customer specific information.

3 Application Layer

The application layer is XML-RPC. This protocol uses an HTTP post to pass a function name and parameters in a standardized XML structure. The function result is returned as an XML document contained in the HTTP response.
See http://www.xml-rpc.com for further information.
The port number on which the service runs, is adjustable in ProgramPasser and defaults to port 5000.

4 RPC API

The API is provided here using pseudo code function prototypes. All types are XML-RPC types and square brackets denote an array of the specified type.

4.1 GetVersion

```
int ProgramPasser.GetVersion()
```

Returns the API verion provided by the server. Current version is **9**. This function is new in version 2, so when calling it returns a 'not implemented' fault, the API version is 1.

4.2 SelectController

```
boolean ProgramPasser.SelectController(int Controller)
```

When multiple carwash controllers are available, they can be accessed through a single program passer interface. Subsequent calls to the API will affect that specific controller. The default controller is 0.
The return value indicates whether the controller was successfully selected.
Added in API level 4.

4.3 AddQueueItem

Adds and item to the program queue.

```
int ProgramPasser.AddQueueItem(
        dateTime.iso8601 TransactionTime,
        string CustomerID,
        string CustomerName,
        int[] Programs
        [, string ProductText,
           int ProductPrice,
           boolean HappyHour,
           int TransactionID,
           int CashRegisterID,
           int CustomerBalance
           [, string LicensePlate
             [, int[] ConditionIDs ] // Added API version 8
           ]
        ]
);
```

The return value indicates the index of the item in the queue after addition. A negative return value indicates that the item could not added:

- -1: No queue available to add the item to.
- -2: Controller doesn't accept the program combination.
- -3: Controller is offline/unreachable.

The customer id and name strings can be empty, but not omitted.
This function was extended in version 2 of the API to have more information available to ProgramPasser. The function arguments that were added are optional, so the function remains backwards compatible.

TransactionID should be a unique number for the transaction. *CashRegisterID* should be a unique number that identifies the cash register.
ProductText should be a descriptive text for chosen wash programs that can be displayed to the customer. *ProductPrice* and *CustomerBalance* represent money in cents.
The *HappyHour* boolean indicates whether it was Happy Hour according to the cash register.
The *LicensePlate* parameter was added in api version 3.
The *ConditionIDs* parameter was added in api version 8 and is intended to accept a list of integers indicating special conditions for this item that can be used by the controller or other external components like CasOut.

4.4 ReplaceQueueItemByID

Replaces an existing queue item with a new one. Added in API level 8.

```
int ProgramPasser.ReplaceQueueItemByID(
        int TransactionID,
        dateTime.iso8601 TransactionTime,
        string CustomerID,
        string CustomerName,
        int[] Programs
        [,string ProductText,
          int ProductPrice,
          boolean HappyHour,
          int NewTransactionID,
          int CashRegisterID,
          int CustomerBalance,
          string LicensePlate,
          int[] ConditionIDs
        ]
);
```

The meaning of the parameters is the same as in AddQueueItem, except that the first parameter is used to identify which item to replace and the original TransactionID will be replaced by NewTransactionID on a successful replacement.
The return value is the (zero based) index of the modified car in the queue or negative in case of failure.

4.5 AttachLicensePlate

Attach a license plate string to existing queue item.

```
boolean AttachLicensePlate(int QueueItemIndex, string LicensePlate)
```

Returns true when the license plate string was successfully added to the queue item. Added in version 3.

4.6 AttachLicensePlateByID

Same as AttachLicensePlate but by transaction ID rather than item index.

```
boolean AttachLicensePlateByID(int TransactionID,
                               string LicensePlate)
```

Added in version 3.

4.7 FetchQueue

Fetch the current queue.

```
struct QueueItem {
  dateTime.iso8601 TransactionTime;
  int TransactionID;
  int CashRegisterID;
  boolean HappyHour;
  string CustomerID;
  string CustomerName;
  string LicensePlate;
  int[] Programs;
  boolean Active;
};

QueueItem[] ProgramPasser.FetchQueue();
```

Returns an array of items waiting in the queue. The *Active* member indicates that the item has already been passed onto the controller, but the controller has not yet accepted the car, so it remains in the queue still. The QueueItem structure was extended in version 2.

4.8 DeleteQueueItem

Deletes an item from the transaction queue.

```
boolean ProgramPasser.DeleteQueueItem(int QueueItemIndex
        [, dateTime.iso8601 TransactionTime]);
```

Returns true when the item was successfully deleted. The optional transaction time can be provided in order to protect an item from being deleted by index when the queue has changed since the last fetch on which the index was based.

4.9 DeleteQueueItemByID

```
boolean ProgramPasser.DeleteQueueItem(int TransactionID);
```

Deletes an item from the queue with specified transaction ID. Return true when an item with specified ID was found in the queue and removed. This function can be used by the cash register to remove and item from the queue when e.g. a transaction was canceled after the associated item was already added to the queue. New in version 2.

4.10 FetchCounters

Fetches wash program counters as registered by ProgramPasser or the connected controller. A time range can be provided for selecting parts of the day. This can be used to select counters during happy hour or regular hour time frames.

```
struct ProgramCounter {
  int ProgramNumber;
  int Counter;
  [boolean HappyHour;]    // May be omitted when not available
  [boolean Option;]       // May be omitted when not available
};

struct ProgramCounters {
  dateTime.iso8601 StartTime;
  dateTime.iso8601 EndTime;
  boolean FromController;
  ProgramCounter[] Counters;
};

ProgramCounters ProgramPasser.FetchCounters([dateTime.iso8601
StartTime[, dateTime.iso8601 EndTime[, boolean FromController =
False]]]);
```

Returns a record containing some flags and an array of wash program counter records. When ProgramPasser has information on happy hour, then the *HappyHour* flag will be set on the counter record and there may be two records for one wash program. The *Option* flag indicates that the wash program is an upsell and not a main wash program.
The *StartTime* and *EndTime* parameters are optional. When not provided they are assumed to be 0:00 at the present day and now, respectively.
The returned *StartTime* and *EndTime* values may not match the ones provided in the request. This can happen when time based selection wasn't possible at the requested resolution. e.g. when hourly counters were requested, but the controller only returns daily counters. The *FromController* flag indicates that the values were fetched from the carwash controller rather than from ProgramPassers own transaction log. This flag may not match the one provided in the request when counters from the controller are not available.

4.11 FetchTransactions

Fetches transactions from the carwash controller. Most carwash controllers don't record this information, so it will be derived from counter logging by ProgramPasser.

```
struct Transaction {
  dateTime.iso8601 Time;
  int ProgramNumber;
  [boolean HappyHour;]    // May be omitted when not available
  [int TransactionID;]    // May be omitted when not available
  [boolean IsOption;]     // Whether the program is an option
                             May be omitted when not available
};

struct TransactionList {
  dateTime.iso8601 StartTime;
  dateTime.iso8601 EndTime;
  Transaction[] Transactions;
};

TransactionList ProgramPasser.FetchTransactions(
    dateTime.iso8601 StartTime,
    dateTime.iso8601 EndTime);
```

Returns a list of transactions. Only a single program is recorded per transaction, so multiple transactions may be listed for a single car when this car has option programs. These transactions will have the same “Time” value. On the other hand, transactions with the same time value are not guaranteed to be associated with a single car. This is due to the nature of the way the transaction data is collected.
Added in API level 5.
Optional TransactionID added to transaction in API level 6.
Optional IsOption added to transaction in API level 7.

4.12 GetGateStatus

```
int ProgramPasser.GetGateStatus()
```

Return status of controller gate. This function is new in API version 9. Returns 0 for open gate (no car) and 1 for closed gate (car in gate). Returns -1 if no gate status is implemented by the controller.

4.13 GetConsumptionIDs

Fetches a list of available consumption monitors. A consumption monitor represent a time registration of some sort of use fluid or material, e.g. of a chemical or water, within the carwash. New in API level 8.

```
struct ConsumptionID {
  string ID;    // Unique identifier for consumption
  string Name;  // Name of particular consumption
  string Unit;  // unit of quantity e.g. L or kg
};

struct ConsumptionIDs {
  ConsumptionID[] IDs;
};

ConsumptionIDs ProgramPasser.GetConsumptionIDs();
```

4.14 GetConsumption

Fetches consumption for the specified time period for the provided ConsumptionID. New in API level 8.

```
struct Consumption {
  string ID;    // Unique identifier for consumption
  dateTime.iso8601 StartTime;
  dateTime.iso8601 EndTime;
  double Value;  // Actual value of consumption over specified period
};

Consumption ProgramPasser.GetConsumption(
    string ConsumptionID,
    dateTime.iso8601 StartTime [, dateTime.iso8601 EndTime]
);
```

5 New style queue functions

Added in API level 9. New queue functions using a “Car” structure rather than individual function arguments. The elements of the “Car” structure are all optional, though obviously not providing “Programs” makes no sense.

```
struct Car {
  dateTime.iso8601 TransactionTime;
  int TransactionID;
  int[] Programs;
  string ProductText;
  string ProductPrice;    // Money string with dot: “1.50”
  boolean HappyHour;
  string CustomerID;
  string CustomerName;
  string CustomerBalance; // Money string with dot: “88.50”
  string LicensePlate;
  int[] ConditionsIDs;    // List of ID's related to the customer e.g.
                          // customer type or special rebates given
  int CashRegisterID;
};
```

5.1 AddCar

Add a car to the queue. Returns the car position in the queue or negative in case of failure. See AddQueueItem.

```
int ProgramPasser.AddCar(
  struct Car car
);
```

5.2 RemoveCar

Remove a car from the queue. Returns the position the car was remove from or negative in case of failure.

```
int ProgramPasser.RemoveCar(
  int TransactionID
);
```

5.3 ReplaceCar

Replace a car in the queue. Returns the position at which the car was replaced or negative in case of failure.

```
int ProgramPasser.ReplaceCar(
  int TransactionID,
  struct Car car
);
```

5.4 CreateCode

Create a (QR)code string for a transaction to be added to the queue by scanning.

```
string ProgramPasser.CreateCode(
  struct Car car
);
```

6 API Examples

6.1 AddQueueItem

Http post request:

```
POST / HTTP/1.0
Content-Type: text/xml
Content-length: 523

<?xml version="1.0"?>
<methodCall>
 <methodName>ProgramPasser.AddQueueItem</methodName>
 <params>
  <param>
   <value>
    <dateTime.iso8601>20091120T16:57:23</dateTime.iso8601>
   </value>
  </param>
  <param>
   <value><string>12345678</string></value>
  </param>
  <param>
   <value><string>Jaap Versteegh</string></value>
  </param>
  <param>
   <value>
     <array>
      <data>
       <value><int>2</int></value>
       <value><int>7</int></value>
      </data>
     </array>
   </value>
  </param>
 </params>
</methodCall>
```

This adds a transaction to the queue with wash program number to and additional upsell wash program number 7. The customer data is provided here, but not required when not available.

Response by ProgramPasser:

```
HTTP/1.1 200 OK
Connection: close
Content-Type: text/xml
Content-Length: 140

<?xml version="1.0"?>
<methodResponse>
 <params>
  <param>
   <value><int>3</int></value>
  </param>
 </params>
</methodResponse>
```

Indicating that the item was inserted at position 3 in the queue, meaning there are 4 cars between the cash register and the carwash tunnel. (queue starts at position 0).

It is **strongly** advised to use an existing and well tested XMLRPC library for your specific development platform, rather than hand coding and parsing the above messages.

7 Barcode Input

7.1 Format

ProgramPasser accepts input from a barcode reader. A wash programming can be added to the queue, using the following format:

```
DDDLTSSSSPUU
```

Each character in this string represents a nibble of a value that is printed in hexadecimal. To prevent easy inspection of the string, each nibble is xor'ed with 0xA before being printed. Meaning of the characters:

```
DDD   : Day of the year. 12 bit unsigned value.
L     : Location number. 4 bit unsigned value.
T     : Tunnel number. 4 bit unsigned value.
SSSS  : 16 bit value.
        Most significant bit indicates happy hour.
        15 remaining bits: Transaction sequential number
P     : Main wash program number. 4 bit unsigned value.
U     : Addition 1 program number. 4 bit unsigned value.
U     : Addition 2 program number. 4 bit unsigned value.
```

7.2 Example

Wash program 2 with addition nr. 11 on the 31st of january, at happy hour, which happened to be the 88th transaction of the day/week/whatever. Location #2, tunnel #1:

```
01F2180582B0 ^ AAAAAAAAAAAA = AB58B2AF281A
```

Note: Not all barcode systems support encoding of characters other than decimal digits, so this barcode may also be printed in decimal format, with the first 8 nibbles as a zero padded uint32 (10 chars) and the remaining 4 nibbles as an uint16 (5 chars)

Abbreviations

API	Application Programming Interface
HTTP	Hypertext Transfer Protocol (http://www.w3.org/Protocols/rfc2616/rfc2616.html)
XML-RPC	eXtensible Markup Language Remote Procedure Calling (http://www.xmlrpc.com)