JSONPOS API

Overview

Poplapay JSONPOS API allows an electronic cash register (ECR) to integrate with the Poplapay payment terminal (PT). This document defines the protocol used between ECR and PT.

Protocol stack

Layers of the protocol stack (bottom up) are:

- Stream layer: (1) a plain TCP connection, (2) a Bluetooth RFCOMM connection, or (3) a USB serial connection between ECR and PT. For TCP, ECR acts as the initiator and connects to TCP port 10001 of the terminal.
- Framing layer: there are two alternatives to message framing: (1) a text based length+message+newline protocol, and (2) WebSocket text messages. Framing and messages are character based and easy to diagnose manually.
- JSON-RPC layer: JSON-RPC methods (request/response) and notifications (request only) are used to implement the payment protocol and connection maintenance such as keepalives. These are described in this document.

General implementation principles

- Ignore any keys not needed for message processing. Tolerate missing keys unless they are truly mandatory (such as required transaction identifiers).
- JSON object keys have no guaranteed ordering. Don't rely on a specific key order (e.g. the order given in examples).
- Use error string codes (such as `TRANSACTION_NOT_FOUND`) as a programmatic identifier for error types. However, avoid depending on specific errors unless really necessary, as the codes may change over time. Specific error codes may be guaranteed for special situations, such as `TRANSACTION_NOT_FOUND` when `Check` cannot locate a matching transaction.
- Ensure JSON-RPC `id` fields sent are unique for each connection, and preferably unique over all connections. The examples in this document use fixed values but actual requests must have unique values.
- Use error `display_message` optional parameter to display error messages for the cashier. The messages may be translated to the `cashier_language` language. If this field is not present, use the main level "message" field instead.

- Be careful when reading data from a socket: there is no guarantee that TCP `read()` calls return data that matches JSON-RPC transport boundaries. A peer should work correctly even if a JSON-RPC message is received in one-byte pieces. Specific error cases to look out for:

    - A `read()` may return a partial message, including a partial length field.
    - A `read()` may return multiple messages at once.
- Avoid dependencies on message timing. There are significant timing differences between devices and software versions. There is no guarantee that a specific message (such as a purchase reponse) is available immediately in a specific place in code.
- Because the protocol is asynchronous, do not rely on a specific response message arriving at a specific place in code. It is always possible that an unrelated notification or keepalive message arrives instead.
- Currency is always required with amount.
- PosMessages are intended to be displayed to the salesperson as such and not intended for concluding the state of the transaction. Availability and sequence of the PosMessages vary by payment methods, language and timing.

Example message formatting

- The method descriptions in this document are pretty-printed JSON messages which include the JSON-RPC fields (such as `method`) but exclude the JSON-RPC transport framing (length, colon, and trailing newline, or WebSocket). Newlines are added for clarity. Long strings like receipt data are truncated in the examples.
- Javascript comments (`//` and `/* comment */`) are used to clarify fields. Comments are not valid JSON and must not appear in actual on-the-wire JSON.
- The JSON-RPC transport framing is not included in the method examples, but is always present for requests, responses, and notifications.
- The `id` fields in the examples are fixed, but an actual implementation must use unique `id` for each request.

The actual on-the-wire JSON format is not pretty printed. Key ordering varies between messages. Comments are not allowed as they are not valid JSON.

JSON-RPC transport summary

Available transport interfaces

Interface	Description
TCP	Plain, unencrypted TCP connection. ECR connects to terminal port 10001. Supports both text-based transport and WebSocket, with autodetection.
Bluetooth RFCOMM	SPm20 only: RFCOMM channel 1, can be discovered via Bluetooth SDP as UUID 00001101-0000-1000-8000-00805F9B34FB. Requires _Sync handshake.
USB serial	SPm20 only (software version 18.5.0 or higher): USB serial. Ensure transparent line discipline (terminal uses and expects LF line endings). Requires _Sync handshake.

Text-based transport

The transport uses a plain TCP, RFCOMM, or USB serial connection which carries JSON-RPC 2.0 framed messages:

- Hex encoded 8-digit length field followed by a colon (`:`).
- JSON-serialized pure ASCII JSON-RPC message (non-ASCII codepoints must be `\uNNNN` escaped).
- Newline (0x0a), not included in the length field.

Example (newline is not shown):

```
00000055:{"jsonrpc":"2.0","method":"ExampleMethod",
"params":{"argument":"value"},"id":"req-1"}
```

Here's a Javascript example of forming a transport framed message:

```
// JSON data must be encoded in ASCII which also means character and byte
// length will match.
function jsonStringifyAscii(val) {
    return JSON.stringify(val).replace(/[\u007f-\uffff]/g, function (x) {
        return '\\u' + ('0000' + x.charCodeAt(0).toString(16)).substr(-4);
    });
}

var id_counter = 0;
var obj = {
    jsonrpc: '2.0',
    method: 'ExampleMethod',
    params: { argument: 'value', nonascii: 'foo\u1234bar' },
    id: 'req-' + (++id_counter)
};

var jsonData = jsonStringifyAscii(obj);
var frame = ('00000000' + jsonData.length.toString(16)).substr(-8) +
            ':' + jsonData + '\n';
```

```
20
21  // Write 'frame' (which is pure ASCII) to the socket.
22  console.log(frame);
```

The result is:

```
1  0000006f:{"jsonrpc":"2.0","method":"ExampleMethod",
2  "params":{"argument":"value","nonascii":"foo\u1234bar"},"id":"req-1"}
```

Messages on the wire are typically one-line packed JSON, but examples below and throughout this document are pretty printed for readability.

A JSON-RPC request has the form:

```
1  // 'method' identifies method to be invoked
2  // 'params' is an object containing method arguments
3  // 'id' is a unique string chosen by the requester (must be unique within
4  //     a single connection, preferably across all connections)
5
6  XXXXXXXX:{
7      "jsonrpc": "2.0",
8      "method": "MethodName",
9      "params": {...},
10     "id": "req-N"
11 }
```

If the request succeeds, the corresponding reply has the form:

```
1  // 'result' is an object containing method results
2
3  XXXXXXXX:{
```

```
    "jsonrpc": "2.0",
    "result": {...},
    "id": "req-N"
}
```

If the request fails, the corresonding error has the form:

```
// 'code' is an integer error code which is ignored
// 'message' provides a short error description (one line)
// 'string_code' provides a string-based error code, e.g. CARD_REMOVED
// 'details' provides an optional traceback or other error details

XXXXXXXX:{
    "jsonrpc": "2.0",
    "error": {
        "code": 1,
        "message": "card removed",
        "data": {
            "string_code": "CARD_REMOVED",
            "details": "CARD_REMOVED: card removed\n\ttraceback...\n\t[...]"
        }
    },
    "id": "req-N"
}
```

A JSON-RPC notification is similar to a method request but lacks an "id" field and doesn't get any reply.

```
XXXXXXXX:{
    "jsonrpc": "2.0",

```

```
    "method": "MethodName",
    "params": {...}
}
```

The JSON-RPC transport defines a few special methods and notifications for connection management:

- `_Keepalive`: request for connection keepalive, respond with empty object, mandatory to implement.
    - However, at present the terminal is lenient and allows a ECR to ignore `_Keepalive` requests: if the ECR never responds to any keepalive requests only one request is sent and its timeout is silently ignored. This is only for backwards compatibility, new implementations must always respond to keepalive requests.
- `_Info`: notification about an informative event, no action required, optional.
- `_Error`: notification about an error, no action required, optional.
- `_CloseReason`: notification about connection being closed, no action required, optional.
- `_Sync`: (re)synchronization of a Bluetooth RFCOMM stream or USB serial stream. Mandatory when using RFCOMM or USB serial.

The JSON-RPC transport is described in more detail in "Common JSON/RPC transport". The `_Sync` command used with RFCOMM is described in this document.

The method descriptions below include JSON-RPC fields but omit the framing.

Local WebSocket transport

An alternative to the `XXXXXXXX:{...}\n` framing is WebSocket:

- WebSocket is autodetected from the same port, 10001. Request URI path is `/jsonpos` so connection URI is `ws://myterminal.local:10001/jsonpos`.
- WebSocket subprotocol is `jsonrpc2.0`.
- **The terminal only supports plain TCP (ws://), not SSL (wss://). As a result, a browser page connecting to the terminal must be served using plain HTTP (not HTTPS) because most browsers don't allow a plain TCP WebSocket connection from a "secure page".**

- JSON-RPC messages are carried as JSON-encoded UTF-8 WebSocket Text messages (opcode 1). WebSocket Binary messages (opcode 2) are not used.
- Terminal doesn't currently send WebSocket Ping messages as they are redundant with `_Keepalive`. ECR must still support them as part of WebSocket. Terminal will respond to Ping messages if sent by ECR.

The code examples below are illustrative only.

To connect to the terminal from a web page:

```
var websock = new WebSocket('ws://myterminal.local:10001/jsonpos',
                            [ 'jsonrpc2.0' ]);
websock.addEventListener('message', function (event) {
    // event.data is a string with the JSON-RPC message; process it.
});
// Other events like 'open', 'close', 'error' are available and should
// be handled.
```

To send a message to the terminal (once connected):

```
websock.send(JSON.stringify({
    jsonrpc: '2.0',
    method: 'Purchase',
    id: getRequestId(),
    params: {
        // ...
    }
}));
```

To close the connection:

```
websock.close(1000, 'closed by ECR');
```

WebSocket only provides a simple JSON message transport. Request/response dispatch must be implemented on top of the raw JSON messaging, just as with the length-message-newline framing.

Remote WebSocket transport

WebSocket can also be used to connect to the terminal via a server endpoint. This works similarly to a local WebSocket connection but uses TLS (wss://) and HTTP Basic authentication. The WebSocket URI path includes a terminal ID (12345 in the example below):

```
// Development:
var websock = new WebSocket(
  'wss://user:pass@api.sandbox.poplatek.com/api/v2/terminal/12345/jsonpos',
  [ 'jsonrpc2.0' ]
);

// Production:
var websock = new WebSocket(
  'wss://user:pass@api.poplatek.com/api/v2/terminal/12345/jsonpos',
  [ 'jsonrpc2.0' ]
);
```

The target terminal is identified using a terminal ID. If multiple terminals are online with the same terminal ID, there is no guarantee which terminal is chosen for the connection.

Some web browsers have problems with handling username and password using HTTP Basic authentication. For these browsers, there's an alternative authentication method using the protocols list, by including the username and password separated by colon encoded in base64url encoding (RFC4648) with `x-popla-auth-` as prefix:

```
var websock = new WebSocket(
  'wss://api.sandbox.poplatek.com/api/v2/terminal/12345/jsonpos',
  [ 'jsonrpc2.0', 'x-popla-auth-' + btoa('user:pass').replace(/\//g, '_').replace(/\+/g, '-').re
);
```

The replacements in the above example are used to transform the base64 encoding produced by `btoa` in to base64url encoding, for example from `YWFhYf/+eg==` to `YWFhYf_-eg`.

The use of this method is meant only as a workaround for browser problems, and not recommended for general usage.

There are minor differences to local WebSocket connections:

- An API key is not required: the HTTP authentication provides authorization.

API key

Most ECR-initiated JSONPOS methods need an `api_key` authenticator given in the request `params` object. An API key is not needed for transport methods and notifications (such as `_Keepalive` and `_CloseReason`), or methods explicitly indicated as being unauthenticated (such as `Status`).

API keys:

- Development: `"api_key": "e086554a-9a35-4bbf-9f99-a8a7cd7d1dcc"`
- Production: request an API key from `developers@poplapay.com`

External data

Some requests like Purchase and Refund may include an `external_data` field containing free form data that the ECR wants to transfer to the payment server for some specific use. Intended usage is for ECR specific identifiers, receipt data, etc. Limitations:

- JSON encoded byte length (shortest encoding without indent etc) must be 50kB (50 * 1024 bytes) or less. Byte length is computed from the UTF-8 encoded form of the encoded JSON data.
- JSON array/object nesting level must be at most 10, with the level including the `external_data` object itself. For example, the following would have a depth of 2: `{ "values": [ 1, 2, 3 ] }`.
- Keys and string values must be valid UTF-8, codepoints in U+D800 to U+DFFF (surrogates) must not be used. Invalid UTF-8 is rejected. Recommendation is to use only `[0-9a-zA-Z_]` in keys.

Identifier spaces

Shared

- `currency`: `EUR`, `GBP`, etc, see ISO 4217. Required when amount is given.

Messages sent by PT

- `transaction_id`: 20-digit number sequence used to identify a transaction for e.g. cancel or refund. PT provides in purchase response, also included in receipt data. The format is opaque in the protocol and currently consist of a 12-digit filing code followed by 8-digit logical terminal ID (on receipts there is a space between the two groups).
- `transaction_unique_id`: Poplapay internal unique identifier for transaction. Identifies a transaction uniquely in Poplapay payment service. Only used for diagnostics and manual investigation.

Messages sent by ECR

- `receipt_id`: Numeric identifier used by ECR to identify a receipt. One receipt may contain multiple purchases.

- `sequence_id`: Numeric identifier used by ECR to identify a transaction uniquely when receipt ID and amount match. Needed only when a receipt contains multiple purchases. Correct currency must always be used.

Offline transactions

Payment terminal supports network offline transactions. During the transaction processing if offline is attempted the payment terminal will send a PosMessage stating that the transaction is about to be performed offline. For example like the message below:

```
{
    "method": "PosMessage",
    "jsonrpc": "2.0",
    "params": {
        "message": "Ongelma verkkoyhteydess\u00e4.\
        Katevarmennuksen vaativat kortit eiv\u00e4t toimi."
    }
}
```

In the transaction response messages, the payment terminal adds a "offline" boolean field indicating whether the transaction was attempted in offline or online.

Methods provided by PT

_Keepalive

Defined in JSON-RPC transport documentation.

Recommended minimum keepalive timeout for ECR-initiated keepalive requests is 5 seconds: while the terminal usually responds in less than a second, there are several cases where the response may take much longer.

_Info

Defined in JSON-RPC transport documentation.

_Error

Defined in JSON-RPC transport documentation.

_CloseReason

Defined in JSON-RPC transport documentation.

_Sync

The `_Sync` message is related to serial-like transports such as RFCOMM and USB serial; it's not used for TCP.

A `_Sync` request brings the JSONPOS connection into synchronized state. Connections delimited by `_Sync` are considered separate logical JSON-RPC connections, analogous to separate TCP connections, but share the same RFCOMM or USB serial link:

- Each logical JSON-RPC connection has a separate request/reply ID space. The same ID can be reused in a new connection and is unrelated to any previous connections.
- Response messages must never be sent to a different logical JSON-RPC connection. When receiving a `_Sync`, pending requests in previous connections must be terminated with an error, as no reply will ever be sent or received for them anymore.

The connection synchronization process is driven by the ECR. The ECR should synchronize (1) when it initially connects to the terminal, (2) when there's a framing parse error or any other reliable error indication, and (3) when `_Keepalive` based monitoring fails.

The (re)synchronization process should be as follows:

- Send a 64-byte filler consisting of newlines (0x0A) before a `_Sync`. Wait ~100ms after the filler has been sent.

    - This is necessary when using SPm20 RFCOMM and suspend/resume. When the SPm20 wakes up from suspended state based on RFCOMM activity, it may lose a few initial bytes and corrupt some bytes after waking up. This filler allows SPm20 suspend/resume to work reliably.
    - If this filler is not sent, an initial `_Sync` sent to SPm20 will typically fail, and a `_Sync` retry will then succeed normally. The filler makes this process faster.
- Optionally, read and consume data until no data has been received for e.g. 1-5 seconds. This reduces the need to discard "garbage" data preceding a `_Sync` reply, but does not eliminate the need for `_Sync` reply scanning altogether.
- Send a properly framed `_Sync` request with a unique "id" field, for example:

```
0000003e:{"jsonrpc":"2.0","method":"_Sync","id":"sync-123","params":{}}<LF>
```

- Scan input stream for a properly properly framed and correctly parsing `_Sync` reply from the connection, e.g.:

```
0000002c:{"jsonrpc":"2.0","id":"sync-123","reply":{}}<LF>
```

  The reply may be preceded by valid JSON-RPC frame(s) related to a previous connection, valid `_Sync` response(s) for a previous sync attempt, or any other garbage data. It is critical to be able to discard such data and keep scanning for the correct, matching reply.

  If the reply parses correctly, be careful to verify that its `id` field matches that of the latest `_Sync` request rather than an old one. The `id` field values should be unique and can be derived e.g. from a timestamp (for example: `"sync-" + Date.now()` in Javascript). If the `id` does not match, the `_Sync` reply must be ignored.

  An actual `_Sync` reply may, like usual, have additional fields.

- If `_Sync` reply parsed correctly the JSON-RPC connection is now functional. The new connection replaces any previous connections, and any pending requests sent via an older connection should be considered failed; the terminal won't be responding to them. In effect a `_Sync` is treated the same as a TCP disconnect followed by a reconnect.
- If a valid `_Sync` reply is not received for e.g. 1-2 seconds, consider the synchronization attempt to have failed, and retry from the start.
- NOTE: The `_Sync` method should only be used with RFCOMM and USB serial, its behavior is unspecified for TCP.

TerminalInfo

Request PT software version information. This request doesn't require an API key.

Request:

```
{
    "jsonrpc": "2.0",
    "method": "TerminalInfo",
    "id": "pos-1",
    "params": {
        // No specific required fields, but any build, software version,
        // or device identification information can be included here.
    }
}
```

Response:

```
{
    "jsonrpc": "2.0",
    "id": "pos-1",
    "result": {
        "hardware_id": "00081927799c",
        "terminal_id": 90300006,
        "version": "14.8.3",
        "revision": "717e9c1",

        // Printer field present only if printer available through API,
        // depending on HW and SW configuration.

```

```
        "printer": {
            "color": "bw",       // black and white
                                 // reserved: "greyscale", "color"
            "max_height": 1000,  // pixels
            "max_width": 385     // pixels
        },

        // Terminal model name.
        "model_name": "SPm20",

        // Both sales location and terminal may have names
        // defined in terminal management tools and API.
        // These can be shown to user or cross checked in
        // ECR implementation to make sure that the ECR
        // is connected to correct terminal running correct
        // configuration.
        "name": "Cashier 2", // Terminal configured name
        "sales_location_name": "Ye Olde Shoppe", // Sales location name

        // Terminal may add arbitrary additional identifiers here.
        // These identifiers may be added and removed without notice.

        // Some terminals may indicate a link speed hint (bytes/second)
        // which is useful for rate limiting.  Currently provided by
        // SPm20 for RFCOMM network proxy use.
        "link_speed": 9000,

        // Optional ECR configuration available for some ECR integrations.
        "ecr_config": {

```

```
            // ECR specific fields
        }
    }
}
```

Current model names:

Name	Description
VEGA3000	Castles V3M2 or V3P3
YOMANI	Worldline YOMANI ML or YOMANI XR
YOXIMO	Worldline YOXIMO
XENOA-ECO	Worldline XENOA ECO
VALINA	Worldline VALINA
SPm20	Spire SPm20

Status

Request PT status. This request doesn't require an API key.

Request:

```
{
    "jsonrpc": "2.0",
    "method": "Status",
    "id": "pos-1",
    "params": {

```

```
            // No specific required fields.
        }
    }
```

The response contains the same fields as a StatusEvent notify, see StatusEvent for details.

Purchase

Request for 123,45 EUR purchase:

```
{
    "jsonrpc": "2.0",
    "method": "Purchase",
    "id": "pos-1",
    "params": {
        "api_key": "e086554a-9a35-4bbf-9f99-a8a7cd7d1dcc",
        "cashier_language": "en",

        // ECR identifiers.
        "receipt_id": 123,                  // mandatory
        "sequence_id": 234,                 // optional

        // Amount and currency.  If amount is missing, only read and
        // return card data (loyalty, tokens).  Currency is mandatory
        // when amount (or cashback_amount) is present.
        "amount": 12345,
        "cashback_amount": 500,  // optional: used only for cashback
                                 // transactions, may not exceed amount
        "currency": "EUR",

```

```
        // Preferred receipt width in characters: optional, e.g. 40.
        // The receipt may contain less characters per line than
        // requested but not more apart from a trailing newline.
        "preferred_receipt_text_width": 40,

        // Forced authorization: ignored at present, terminal always
        // forces authorization.
        "forced_authorization": true,

        // Optional features, enabled only on selected terminals.

        // Option to stop processing on card insert/swipe/tap. Terminal stops
        // processing and queries ECR how to proceed with CardInfo request.
        "stop_on_card_info": false,


        // Request a CardInfo call when card inserted
        "request_card_info": {
            // Set 'scan_all_apps_for_info' to true to get non_payment_data
            // for all chip applications for loyalty programs etc.
            "scan_all_apps_for_info": false
        },

        // Request a AppInfo call when card inserted
        "request_app_info": {
            // See request_card_info for parameter fields
        },
```

```
        // Request card information to successful purchase response
        "request_result": {
            // Tokens for the card the purchase was made with; or in error
            // may give the last card that was processed
            // Set 'scan_all_apps_for_info' to true to get non_payment_data
            // for all chip applications for loyalty programs etc.
            "scan_all_apps_for_info": false

        },

        // If present (and true), the request does not time out.
        // After first card interaction the transaction may time out.
        "no_timeout": false,

        // Optional free form external data object.
        "external_data": {
            "name": "John Doe",
            "shift": {
                "number": 123
            }
        }
    }
}
```

Success response:

```
{
    "jsonrpc": "2.0",
    "id": "pos-1",

```

```
    "result": {
        "offline": false,             // optional for online transactions,
                                      // default to false
        "amount": 12345,
        "currency": "EUR",
        "transaction_id": "14101500003890300006",
        "transaction_unique_id": "70ca6015-8466-4678-9557-beb87ae390bd",
        "forced_authorization": false,
        "card_reading_method": "chip",
        "payment_method": "credit",
        "debit_credit_certain": true,
        "response_text": "Approved",
        "response_code": "00",

        // Primary account number, masked for salesperson.
        "pan_masked_for_clerk": "527591******3684",

        // Primary account number, masked for use in customer receipts.
        "pan_masked_for_customer": "************3684",

        // Sequence number for primary account number, optional
        "pan_sequence_number": "01",

        // Application name for receipt.
        "card_name": "Debit MasterCard",

        // Transaction time for receipt
        "transaction_time": "160517115724",


```

```
        // Authorization code returned from issuer.
        // May be missing or "000000" for offline transactions
        "authorization_code": "123XYZ",

        // Application cryptogram, for chip transactions
        "application_cryptogram": "30A562A9E7E99DC9"

        // Application identifier (AID), for chip transactions
        "application_id": "A0000000041010",

        // Receipt text.
        "merchant_receipt": {
            "signature_required": false,
            "id_check_required": false,
            "text": "Selite: Veloitus 1,07 EUR\nKortti: VISA\nNumero:..."
        },
        "customer_receipt": {
            "text": "Selite: Veloitus 1,07 EUR\nKortti: VISA\nNumero:..."
        },

        // Merchant number for the settlement contract selected for
        // the payment card application.
        "merchant_number": "09388984",

        // List of valid schemes for the selected contract.
        "contract_valid_for_schemes": [
            "VI",
            "MC"
        ],

```

```
65
66          // Optional non-payment card data, such as magnetic stripe data
67          // for non-payment cards, or detected loyalty data.
68          // List of entries. See chapter Non-payment-data for description.
            "non_payment_data": []
        }
    }
```

Error response:

```
{
    "jsonrpc": "2.0",
    "id": "pos-1",
    "error": {
        "code": 1,
        "message": "card removed during emv pin entry operation",
        "data": {
            "string_code": "CARD_REMOVED",
            "display_message": "...",
            "details": "Error: CARD_REMOVED: card removed during pin entry...",

            "offline": false,
            "amount": 12345,
            "currency": "EUR",
            "transaction_id": "",
            "transaction_unique_id": "70ca6015-8466-4678-9557-beb87ae390bd",
            "forced_authorization": false,
            "card_reading_method": "chip",
            "payment_method": "credit",

```

```
            "debit_credit_certain": true,
            "response_text": "Declined",
            "response_code": "06",
            "pan_masked_for_clerk": "527591******3684",
            "pan_masked_for_customer": "************3684",
            "card_name": "Debit MasterCard",
            "transaction_time": "160517115724",
            "merchant_receipt": {
                "signature_required": false,
                "id_check_required": false,
                "text": "Selite: Ei veloitusta 1,07 EUR\nKortti: VISA\nNumero:..."
            },
            "customer_receipt": {
                "text": "Selite: Ei veloitusta 1,07 EUR\nKortti: VISA\nNumero:..."
            },

            // See success example.
            "store_token": "PIT1234567890123456789",
            "lookup_tokens": ["PIT1234567890123456789", "PIT1644567490123456989"],

            // In case tokenization has failed.
            "tokenization_error_code": "INTERNAL_ERROR",
            "tokenization_error_description": "Error: INTERNAL_ERROR: Token...",
            "tokenization_error_details": "Error: INTERNAL_ERROR: Token..."
        }
    }
}
```

Depending on the error type, error receipt data, masked PAN, `card_name` and `transaction_time` may or may not be present. For example, if purchase request is missing the amount field the result might be for example:

```
{
    "jsonrpc": "2.0",
    "id": "pos-1",
    "error": {
        "code": 1,
        "message": "/app/lib/usecase/jsonpos_base.lua:77: assertion failed!",
        "data": {
            "string_code": "UNKNOWN",
            "display_message": "...",
            "details": "Error: UNKNOWN: /app/lib/usecase/jsonpos_base.lua:77:..."
        }
    }
}
```

Refund

Request example:

```
{
    "jsonrpc": "2.0",
    "method": "Refund",
    "id": "pos-1",
    "params": {
        "api_key": "e086554a-9a35-4bbf-9f99-a8a7cd7d1dcc",
        "cashier_language": "en",

```

```
        // ECR identifiers.
        "receipt_id": 124,  // mandatory, for Refund (does not match orig Purchase)
        "sequence_id": 235, // optional, for Refund (does not match orig Purchase)

        // Mandatory amount, up to original Purchase amount.
        // Currency is mandatory and must match Purchase.
        "amount": 45,
        "currency": "EUR",

        // Optional transaction ID.
        // If present, must match Purchase from same sales location.
        // If present, Purchase transaction must not be older than the transaction
        // storage period in payment gateway.
        "transaction_id": "14101500003890300006",

        // Optional, use together with transaction_id to enforce refund
        // to be done with same card as original transaction
        "check_same_card": true,

        // Optional receipt width, see Purchase description.
        "preferred_receipt_text_width": 40,

        // Optional free form external data object.
        "external_data": {
            "name": "John Doe",
            "shift": {
              "number": 123
            }
        }
```

```
    }
}
```

Response, succesfully refunded:

```
{
    "jsonrpc": "2.0",
    "id": "pos-1",
    "result": {
        "offline": false,
        "amount": 12345,
        "currency": "EUR",
        "transaction_id": "14120200000190300005",
        "transaction_unique_id": "b06681ac-5243-4998-837b-0f648913a662",
        "card_reading_method": "chip",
        "payment_method": "debit",
        "debit_credit_certain": true,
        "response_code": "00",
        "response_text": "Approved",
        "pan_masked_for_clerk": "527591******3684",
        "pan_masked_for_customer": "************3684",
        "pan_sequence_number": "01",
        "card_name": "Debit MasterCard",
        "transaction_time": "160517115724",
        "application_cryptogram": "30A562A9E7E99DC9"
        "application_id": "A0000000041010",

        "customer_receipt": {
            "clerk_signature_required": true,

```

```
            "text": "...",
        },
        "merchant_receipt": {
            "id_check_required": false,
            "signature_required": false,
            "text": "..."
        },

        "merchant_number": "09388984",
        "contract_valid_for_schemes": [
          "VI",
          "MC"
        ]
    }
}
```

Response, corresponding purchase not found:

```
{
    "id": "pos-1",
    "jsonrpc": "2.0",
    "error": {
        "code": 1,
        "message": "original protocol transaction not found",
        "data": {
            "offline": false,
            "amount": 345,
            "currency": "EUR",
            "transaction_id": "",

```

```
            "transaction_unique_id": "b06681ac-5243-4998-837b-0f648913a662",
            "card_reading_method": "chip",
            "payment_method": "debit",
            "debit_credit_certain": true,
            "response_code": "06",
            "response_text": "Declined",

            "customer_receipt": {
                "clerk_signature_required": false,
                "text": "..."
            },
            "details": "Error: PROTOCOL_TRANSACTION_UNKNOWN...",
            "merchant_receipt": {
                "id_check_required": false,
                "signature_required": false,
                "text": "...",
            },
            "string_code": "PROTOCOL_TRANSACTION_UNKNOWN",
            "display_message": "...",
        }
    }
}
```

Cancel

Cancellation causes the authorization of a previous purchase to be cancelled and avoids debiting the purchase. If the purchase has already been settled, cancellation will fail.

Offline cancel is currently not supported.

Request to cancel a previous purchase; `amount`, `currency`, and `transaction_id` must all match:

```
{
    "jsonrpc": "2.0",
    "method": "Cancel",
    "id": "pos-1",
    "params": {
        "api_key": "e086554a-9a35-4bbf-9f99-a8a7cd7d1dcc",
        "cashier_language": "en",

        // Mandatory amount and currency.
        "amount": 12345,
        "currency": "EUR",

        // Mandatory transaction ID, from Purchase response
        // or receipt.
        "transaction_id": "14101500003890300006"
    }
}
```

Response when transaction is found, not yet cancelled, and was successfully cancelled:

```
{
    "id": "pos-1",
    "jsonrpc": "2.0",
    "result": {
        "offline": false,
        "result": true


```

```
        }
    }
```

Response when transaction is found but is already cancelled:

```
{
    "id": "pos-1",
    "jsonrpc": "2.0",
    "result": {
        "result": true,
        "already_cancelled": true
    }
}
```

Response in other error cases (e.g. `transaction_id` is not valid) uses normal error mechanism:

```
{
    "id": "pos-1",
    "jsonrpc": "2.0",
    "error": {
        "code": 1,
        "message": "protocol transaction not found",
        "data": {
            "string_code": "PROTOCOL_TRANSACTION_UNKNOWN",
            "display_message": "...",
            "details": "Error: PROTOCOL_TRANSACTION_UNKNOWN: protocol..."
        }
    }
}
```

Abort

Abort an ongoing operation (Purchase or Cancel):

```
{
    "jsonrpc": "2.0",
    "method": "Abort",
    "id": "pos-1",
    "params": {
        "api_key": "e086554a-9a35-4bbf-9f99-a8a7cd7d1dcc",

        // Optional field to disable cardholder abort notification if set to
        // true.
        "silent": false
    }
}
```

If an operation is ongoing, responds when the ongoing operation has been completed or terminated (ECR can issue a new operation right after Abort response arrives):

```
{
    "id": "pos-1",
    "jsonrpc": "2.0",
    "result": {
        // no specific results
    }
}
```

If no operation is active but a card is in the card reader, PT will prompt for card removal and respond with the following when card has been removed:

```
{
    "id": "pos-1",
    "jsonrpc": "2.0",
    "result": {
        // no specific results
    }
}
```

If no operation is active and no card is present a `NO_ACTIVE_TRANSACTION` error is sent in response:

```
{
    "id": "pos-1",
    "jsonrpc": "2.0",
    "error": {
        "code": 1,
        "message": "no active transaction",
        "data": {
            "string_code": "NO_ACTIVE_TRANSACTION",
            "display_message": "...",
            "details": "Error: NO_ACTIVE_TRANSACTION: no active transaction..."
        }
    }
}
```

Finally, some internal errors may happen (as with any request):

```
{
    "id": "pos-1",
    "jsonrpc": "2.0",
    "error": {
        "code": 1,
        "message": "no open transaction",
        "data": {
            "string_code": "UNKNOWN",
            "display_message": "...",
            "details": "Error: UNKNOWN: internal error..."
        }
    }
}
```

Check

Check the status of an on-going or already completed transaction made with this particular PT. The transaction is searched from a short transaction history maintained by the terminal (e.g. 10 entries). Amount, currency, receipt ID, and sequence ID are used to make sure the correct transaction is matched. (`transaction_id` is not used by `Check` because it is not available when a transaction is possibly pending.)

Request example:

```
{
    "jsonrpc": "2.0",
    "method": "Check",
    "id": "pos-1",
    "params": {
        "api_key": "e086554a-9a35-4bbf-9f99-a8a7cd7d1dcc",

```

```
        "amount": 12345,            // mandatory
        "currency": "EUR",          // mandatory
        "receipt_id": 123,          // mandatory
        "sequence_id": 234          // recommended
    }
}
```

If sequence number is not given, the latest transaction where other fields match is selected.

If the purchase is still active:

```
{
    "id": "pos-1",
    "jsonrpc": "2.0",
    "result": {
        "terminal_processing": true
    }
}
```

If the purchase has completed, the response is the same that a Purchase result would be, e.g. here the purchase was declined:

```
{
    "id": "pos-1",
    "jsonrpc": "2.0",
    "error": {
        "code": 1,
        "message": "/usr/local/lib/lua/5.1/future.lua:121: timed out...",
        "data": {
            "string_code": "UNKNOWN",
            "display_message": "...",
```

```
    "details": "Error: UNKNOWN: /usr/local/lib/lua/5.1/future.lua:121:...",

    "offline": false,
    "amount": 107,
    "currency": "EUR",
    "transaction_id": "",
    "transaction_unique_id": "8c688b80-a563-4d0f-bf73-9fb78a051d83",
    "forced_authorization": false,
    "payment_method": "credit",
    "debit_credit_certain": false,
    "response_code": "06",
    "response_text": "Declined",

    "merchant_receipt": {
        "signature_required": false,
        "id_check_required": false,
        "text": "Selite: Ei veloitusta 1,07 EUR\nKortti: \n..."
    },
    "customer_receipt": {
        "text": "Selite: Ei veloitusta 1,07 EUR\nKortti: \n..."
    },
    "merchant_number": "09388984",
    "contract_valid_for_schemes": [
      "VI",
      "MC"
    ],

    "parameter_uuid": "01J7NH3GHFMQGM0DBBJPG9TEKH",
    "parameter_set_id": "01J7NH3YEHJNBKDKZ4DRVJMCCJ"

```

```
        }
    }
}
```

If the purchase cannot be located at all, the error code `TRANSACTION_NOT_FOUND` is used:

```
{
    "id": "pos-1",
    "jsonrpc": "2.0",
    "error": {
        "code": 1,
        "message": "transaction not found",
        "data": {
            "string_code": "TRANSACTION_NOT_FOUND",
            "display_message": "...",
            "details": "Error: TRANSACTION_NOT_FOUND: transaction not found..."
        }
    }
}
```

As with other requests, other errors can also occur.

Print

Request the terminal to print the given bitmap. The maximum size of bitmap is given in TerminalInfo response (e.g. width 385, height 1000).

Longer prints may be done with several calls, with eject set to false on all but last fragment.

Request example:

```
{
    "jsonrpc": "2.0",
    "method": "Print",
    "id": "pos-1",
    "params": {
        "api_key": "e086554a-9a35-4bbf-9f99-a8a7cd7d1dcc",

        // Feed paper after printing image.
        // Optional, default = true.
        "eject": true,

        // Allow print operation to show screens and require user input.
        // Optional, default is false for locally initiated Print, true for
        // remote WebSocket transport.
        "interactive": false,

        // Image data.
        "image": {
            // Format of data, currently "png".
            "format": "png",

            // Data as a base-64 encoded PNG bitmap.
            "data": "iVBORw0KGgoAAAANSUhEUgAAAW0AAADICAIAAACsxSecAAAACXBIWXMAAA\
            sTAAALEwEAmpwYAAAAB3RJTUUH3woPBx8Rn8o0RAAAABl0RVh0Q29tbWVudABDcmVhd\
            GVkIHdpdGggR0lNUFeBDhcAAAHUSURBVHja7dSxDQAgDAPBhP13QkAgDAPBhP13NiOkQ\
            AAAAAAAAAAAAAAAAAA8KoeF0ncBF9noodQLB8Bh3QE0BFARwAdAXQEQEcAHQF0BNA\
            RAB0BdATQEUBHAHQE0BFARwAdAdARQEcAHQF0BEBHAB0BdATQEQAdAXQE0BFARwB0BN\
            ARQEcAHQHQEUBHAB0BdARARwAdAXQE0BEAHQF0BNARQEcAdATQEUBHAB0B0BFARwAdA\
            XQEQEcAHQF0BNARAB0BdATQEUBHAHQE0BFARwAdAdARQEcAHQF0BEBHAB0BdATQEQAd\
```

```
            AXQE0BFARwB0BNARQEcAHQHQEUBHAB0BdARARwAdAXQE0BEAHQF0BNARQEcAdATQEUB\
            HAB0B0BFARwAdAXQEQEcAHQF0BNARAB0BdATQEUBHAHQE0BFARwAdAdARQEcAHQF0BE\
            BHAB0BdATQEQAdAXQE0BFARwB0BNARQEcAHQHQEUBHAB0BdATQEQAdAXQE0BFARwB0B\
            NARQEcAHQHQEUBHAB0BdARARwAdAXQE0BEAHQF0BNARQEcAdATQEUBHAB0B0BFARwAd\
            AXQEQEcAHQF0BNARAB0BdAQAAAAAAAAAAAAAAAAAAAuGMDAjUEVLwTKcgAAAAASUV\
            ORK5CYII="
        }
    }
}
```

Success response: print job has been successfully queued.

```
{
    "jsonrpc": "2.0",
    "id": "pos-1",
    "result": {
        "result": true
    }
}
```

DisplayScreen

Request the terminal to display a screen. Abort method can be used to dismiss the screen. For generic screens see "Screen display with DisplayScreen method".

```
{
    "jsonrpc": "2.0",
    "method": "DisplayScreen",
    "id": "pos-1",
```

```
    "params": {
        "api_key": "e086554a-9a35-4bbf-9f99-a8a7cd7d1dcc",
        "cashier_language": "en",

        // Arguments for the screen.
        "scr_args": {
            // Mandatory screen ID, must be white listed in payment
            // terminal parameters to be shown.
            "scr_id": "my_screen"

            // Additional screen specific arguments may be included.
        }
    }
}
```

Success response: Input screen successfully displayed, user chose the "accept" action and a screen related value (e.g. entry field) was "111":

```
{
    "jsonrpc": "2.0",
    "id": "pos-1",
    "result": {
        "action": "accept",
        "value": "111"
    }
}
```

Success response: Input screen successfully displayed and user canceled:

```
{
    "jsonrpc": "2.0",
    "id": "pos-1",
    "result": {
        "action": "cancel"
    }
}
```

Error response: screen not white listed, `BAD_CONFIG` error is returned (also other errors possible):

```
{
    "id": "pos-1",
    "jsonrpc": "2.0",
    "error": {
        "code": 1,
        "message": "scr_id not white listed",
        "data": {
            "string_code": "BAD_CONFIG",
            "display_message": "...",
            "details": "Error: BAD_CONFIG: scr_id not white listed..."
        }
    }
}
```

DisplayScreen is generally not available during transaction processing. DisplayScreen is, however, allowed during CardInfo and AppInfo requests to, for example, request user input. It is also possible to display notifications without waiting for DisplayScreen response. In this case a screen later displayed by the terminal will cause the previous DisplayScreen to fail with a "screen replaced" error.

For example:

```
{
    "jsonrpc": "2.0",
    "method": "DisplayScreen",
    "id": "pos-1",
    "params": {
        "api_key": "e086554a-9a35-4bbf-9f99-a8a7cd7d1dcc",
        "cashier_language": "en",
        "scr_id": "generic_info",
        "scr_args": {
          "text1": "Customer identified.",
          "text2": "Payment with",
          "text3": "loyalty discount"
        },
        // Minimum time (sec) before screen may be replaced with another screen.
        "min_time": 2
    }
}
```

Beep

Request terminal to beep using the default "attention" beep. Beep capabilities are terminal dependent, for example on SPm20 the default attention beep is (currently) a series of three short beeps (beep-beep-beep). There's no control over the beep frequency, duration, or volume yet.

Request example:

```
{
    "jsonrpc": "2.0",
    "method": "Beep",
    "id": "pos-1",
```

```
    "params": {
        "api_key": "e086554a-9a35-4bbf-9f99-a8a7cd7d1dcc"
    }
}
```

Success response:

```
{
    "jsonrpc": "2.0",
    "id": "pos-1",
    "result": {
        // no result fields
    }
}
```

Reboot

Request terminal to reboot as soon as possible, with optional reason string. The method responds immediately, but the reboot may be delayed e.g. if a Purchase is in progress.

Request example:

```
{
    "jsonrpc": "2.0",
    "method": "Reboot",
    "id": "pos-1",
    "params": {
        "api_key": "e086554a-9a35-4bbf-9f99-a8a7cd7d1dcc",

```

```
        // Optional reason.
        "reason": "ECR force reboot button clicked"
    }
}
```

Success response:

```
{
    "jsonrpc": "2.0",
    "id": "pos-1",
    "result": {
        // no result fields
    }
}
```

NetworkStart

Indicate that ECR supports network connection proxying and requests network connections to be handled via the ECR. Connection proxying can be stopped using `NetworkStop` and is automatically stopped if the JSONPOS connection is lost.

API key is not required.

Request example:

```
{
    "jsonrpc": "2.0",
    "method": "NetworkStart",
    "id": "pos-1",
    "params": {
        // no parameters
```

```
    }
}
```

Success response:

```
{
    "jsonrpc": "2.0",
    "id": "pos-1",
    "result": {
        // no result fields
    }
}
```

NetworkStop

Indicate that previously started network proxying should stop. This won't affect ongoing connections opened if network proxying was previously allowed. Network proxying stops automatically if the JSONPOS connection is lost. This method is typically not needed in practice.

API key is not required.

Request example:

```
{
    "jsonrpc": "2.0",
    "method": "NetworkStop",
    "id": "pos-1",
    "params": {
        // no parameters
    }
}
```

Success response:

```
{
    "jsonrpc": "2.0",
    "id": "pos-1",
    "result": {
        // no result fields
    }
}
```

NetworkDisconnected

Sent by ECR to indicate that a connection has been disconnected, either by an explicit request from PT or because a remote peer closed the connection.

The ECR must not send a NetworkDisconnected if there's pending data not yet transferred to the terminal. Data may easily queue up due to rate limiting of an RFCOMM connection, and sending a NetworkDisconnected too early causes some data delivered later to be ignored by the terminal.

To ensure terminal state related to a connection is robustly released, the ECR is allowed to send this method multiple times. The terminal may either ignore a duplicate `NetworkDisconnected` or send back an error indicating the connection ID is no longer recognized (which the ECR must then ignore).

API key is not required.

Request example:

```
{
    "jsonrpc": "2.0",
    "method": "NetworkDisconnected",
    "id": "pos-1",

```

```
    "params": {
        "connection_id": 12,

        // Optional reason, e.g. "requested by pt",
        // "remote peer closed connection", etc.
        "reason": "free form reason"
    }
}
```

Success response:

```
{
    "jsonrpc": "2.0",
    "id": "pos-1",
    "result": {
        // no result fields
    }
}
```

Data

Plain TCP data received from the Internet for a certain logical connection. The `data` property contains base-64 encoded data. The base-64 format has no spaces or newlines, but does have padding characters (=).

Recommended maximum data size per request is 1024 raw bytes which expands to about 1.4kB of JSONPOS data (size limit must not be assumed by receiver). Sending of Data notifys must be rate limited when using RFCOMM to ensure that e.g. `_Keepalive` monitoring won't be disturbed. Fairness across connections must be guaranteed e.g. by sending data for active connections in a round-robin fashion.

If the connection ID is unknown the Data notify must be ignored. If the connection ID is known but the connection state doesn't allow data traffic at present (e.g. connection is not yet ready) the data notify should be dropped and the proxied connection should be aborted (if not already

disconnected).

To minimize message size:

- The `Data` method is a notification (not a request). The method name is short on purpose (Data instead of NetworkData).
- The `connection_id` field is named `id` for `Data`.
- API key is not required for `Data`.

Request (notify) example:

```
{
    "method": "Data",
    "jsonrpc": "2.0",
    "params": {
        "id": 12,
        "data": "<base64 bytes>"
    }
}
```

The terminal is required to process `Data` notifys in sequence to ensure they're processed in the order ECR sends them.

Test

Run a development-time test identified by `test_id`. Individual tests may be added and removed without notice. Tests are documented separately.

```
{
    "jsonrpc": "2.0",
    "method": "Test",
    "id": "pos-1",
    "params": {

```

```
        "api_key": "e086554a-9a35-4bbf-9f99-a8a7cd7d1dcc",

        // Test identifier is mandatory, other parameters may be required
        // depending on the test.
        "test_id": "large_file_download"
    }
}
```

Success response:

```
{
    "jsonrpc": "2.0",
    "id": "pos-1",
    "result": {
        // Result fields vary.
    }
}
```

Methods provided by ECR

_Keepalive

Defined in JSON-RPC transport documentation.

_Info

Defined in JSON-RPC transport documentation.

_Error

Defined in JSON-RPC transport documentation.

_CloseReason

Defined in JSON-RPC transport documentation.

Reconnect

Sent when the terminal wants to close the JSONPOS connection. When received, ECR should finish pending operation(s) such as purchases or refunds, not initiate new requests, and close the connection as soon as possible. ECR can then initiate a new connection. An optional `reason` field indicates a diagnostic reason which can be logged but must not be used programmatically.

If ECR doesn't implement the method, an error must be sent back. The terminal will then close the connection using a best effort approach with no specific guarantees, e.g. when no operations are active.

Example:

```
{
    "jsonrpc": "2.0",
    "method": "Reconnect",
    "id": "pos-1",
    "params": {
        "reason": "installing updates"
    }
}
```

CardInfo

Message sent to provide card info available after card insert/swipe/tap before cardholder application selection. Sent if `stop_on_card_info` flag was enabled for purchase.

```
{
    "jsonrpc": "2.0",
    "method": "CardInfo",
    "id": "pos-1",
    "params": {
        // ECR identifiers.
        "receipt_id": 123,   // mandatory, same as in Purchase message
        "sequence_id": 234,  // optional, same as in Purchase message

        // Card info.
        "card_reading_method": "chip",

        // Language used to display cardholder screens
        "cardholder_language": "es",

        // Non-payment card data, such as magnetic stripe data
        // for non-payment cards, or detected loyalty data.
        // List of entries. See chapter Non-payment-data for description.
        "non_payment_data": [],

        // If loyalty enabled: type of loyalty program detected on
        // card (or missing if not loyalty detected):
        "loyalty_type": "s_etu",

        // If loyalty enabled: loyalty program identifier (if any),
        // format depends on loyalty program.

```

```
        "loyalty_id": "12345",

        // For chip cards: If token generation for all applications enabled,
        // contains masked PAN for each application, otherwise not present.
        // For magstripe and contactless: If token generation enabled,
        // masked PAN of the selected contactless application or the magstripe.

        "card_applications": [
          {
            "pan_masked_for_clerk": "527591******3684",
            "pan_masked_for_customer": "************3684"
          },
          {
            "pan_masked_for_clerk": "527591******8721",
            "pan_masked_for_customer": "************8721"
          }
        ]
    }
}
```

Response:

```
{
    "jsonrpc": "2.0",
    "id": "pos-1",
    "result": {
        // Mandatory action:
        //  - "continue": continue the purchase
        //  - "change_amount": change the amount for this purchase

```

```
        //  - "end": stop the purchase
        // Error response, invalid response, or timeout handled
        // as 'end'.
        "action": "continue",

        // Updated amount and currency, present if and only if
        // action is "change_amount".  "cashback_amount" is optional
        // and may not exceed amount.
        "amount": 12345,
        "cashback_amount": 500,
        "currency": "EUR"
    }
}
```

AppInfo

Message sent to provide card info available after application selection. Send if `stop_on_loyalty` flag was enabled for purchase, and a loyalty program was detected or if `stop_on_appinfo` flag was enabled.

```
{
    "jsonrpc": "2.0",
    "method": "AppInfo",
    "id": "pos-1",
    "params": {
        // ECR identifiers.
        "receipt_id": 123,   // mandatory, same as in Purchase message
        "sequence_id": 234,  // optional, same as in Purchase message

        // Card info.
```

```
        "card_reading_method": "chip",

        // Language used to display cardholder screens
        "cardholder_language": "es",

        // Application info (see Purchase for details).
        "pan_masked_for_clerk": "527591******3684",
        "pan_masked_for_customer": "************3684",
        "card_name": "Debit MasterCard",

        // Non-payment card data, such as magnetic stripe data
        // for non-payment cards, or detected loyalty data.
        // List of entries. See chapter Non-payment-data for description.
        "non_payment_data": [],

        // If loyalty enabled: type of loyalty program detected on
        // card (or missing if not loyalty detected):
        "loyalty_type": "s_etu",

        // If loyalty enabled: loyalty program identifier (if any),
        // format depends on loyalty program.
        "loyalty_id": "12345"
    }
}
```

Response:

```
{
    "jsonrpc": "2.0",

```

```
    "id": "pos-1",
    "result": {
        // Mandatory action:
        //  - "continue": continue the purchase
        //  - "change_amount": change the amount for this purchase
        //  - "end": stop the purchase
        // Error response, invalid response, or timeout handled
        // as 'end'.
        "action": "continue",

        // Updated amount and currency, present if and only if
        // action is "change_amount".  "cashback_amount" is optional
        // and may not exceed amount.
        "amount": 12345,
        "cashback_amount": 500,
        "currency": "EUR"
    }
}
```

PosMessage

A `PosMessage` notification is sent by PT as the purchase sequence proceeds so that a salesperson can easily see what action is expected of the user. The notification contains a human readable message which can require e.g. a card to be inserted. The message is intended for a salesperson. Ignore in unattended environments.

`PosMessage` is human readable text only. **The specific message may change, use different languages, and MUST NOT be parsed programmatically e.g. to detect transaction state as any such logic would be fragile.**

Request (notify) example:

```
{
    "method": "PosMessage",
    "jsonrpc": "2.0",
    "params": {
        "message": "Laita/ved\u00e4 kortti"
    }
}
```

Note that the request has no `id` field because this is a notification message which needs no response.

StatusEvent

A notification about status changes in the terminal. Message is delivered when value for some status field changes. All intermediate status transitions are NOT guaranteed to generate a new notification.

Unless explicitly mentioned, status keys/values may change in terminal versions. All ECR integration to status fields should be soft in nature, i.e. tolerate missing fields and changes in field types or values.

```
{
    "method": "StatusEvent",
    "jsonrpc": "2.0",
    "params": {
        // UTC timestamp indicating when StatusEvent was created.  ETA
        // values (like "update_eta") are relative to this timestamp.
        "timestamp": "2018-04-11T13:23:53.000Z",

        // Indicates whether PT is ready for starting a transaction from a payments
        // perspective, i.e. connection to PSP works or offline transactions are
        // allowed and offline limits have not been reached.  An ongoing transaction

```

```
        // does not cause this flag to go false.
        "ready_for_transaction": true,

        // Indicates whether PSP connection is available or not.
        // The value gets updated with some delay when there is
        // a network outage or a service break.
        "psp_connection_available": true,

        // Status of an ongoing transaction.
        "transaction_status": "PROCESSING",

        // Chip card insertion status. Field is present only when chip card is
        // in contact chip reader.
        "chip_card_in": true,

        // Update status fields.  Update status is one of: PREPARING, CHECKING,
        // DOWNLOADING, or PENDING; progress (percentage) and ETA (seconds)
        // fields are valid in DOWNLOADING state.  If no update check is in
        // progress, the fields are absent.
        "update_status": "DOWNLOADING",
        "update_progress": 63,
        "update_eta": 203,

        // Battery info (currently for SPm20, V3M2, V3P3).
        "battery_percentage": 94,
        "battery_charging": true,
        "plugged_in": true

        // Terminal may add arbitrary additional keys in future

```

```
            // versions.
        }
    }
```

Field `transaction_status` is available if there is an on-going transaction. Currently possible values are:

Status	Description
PROCESSING	Terminal is processing the transaction. For example, chip communication or authorization is going on.
WAIT_CARD_IN	Terminal is waiting for cardholder to present a card.
WAIT_CARD_OUT	Terminal is waiting for cardholder to remove the card from the terminal.
WAIT_POS	Terminal is waiting for a ECR response, e.g. to a CardInfo or an AppInfo request.

Other `transaction_status` values may be added in the future. The ECR should treat any unrecognized new codes the same as PROCESSING. This status field may be used for example user guidance but must not be used for determining if transaction has been completed (use Purchase or Check response instead).

SwipeEvent

A notification about magnetic card swipe without on-going Purchase, Refund or Cancel request.

```
{
    "method": "SwipeEvent",
    "jsonrpc": "2.0",
    "params": {
        // Optional non-payment card data, such as magnetic stripe data
        // for non-payment cards, or detected loyalty data.
        // List of entries. See chapter Non-payment-data for description.
```

```
        "non_payment_data": []
    }
}
```

NetworkConnect

Request for a new plain TCP connection using a hostname and a port.

Terminal provides a numeric connection ID which associates all messages and notifys related to the connection with one another. The connection ID is an arbitrary (not necessarily sequential) number; the ECR should make no assumptions about the connection ID except that it is currently guaranteed to be an unsigned 32-bit value. The terminal may reuse a connection number once all state related to it has been cleared.

Because the client chooses the connection ID, the client can send a `NetworkDisconnect` to abort a pending connection attempt before a NetworkConnect reply has been sent. The ECR must reject with error any network related method calls for a connection ID it has no state for.

The ECR must impose a reasonable timeout for the connection attempt (e.g. 10 seconds) so that a `NetworkConnect` eventually gets a response. The terminal drives retries on its own, so it's enough for the ECR to try the connection once using e.g. a native socket `connect()` call.

There may be multiple active connections at the same time. The connections are identified and kept separate using the connection ID.

NOTE: All established connections must be released if the underlying JSON-RPC transport is closed for any reason. This includes `_Sync` requests.

Request example:

```
{
    "method": "NetworkConnect",
    "jsonrpc": "2.0",
    "id": "pt-1",
    "params": {
        "connection_id": 12,
        "host": "pt.api.sandbox.poplatek.com",
```

```
        "port": 443
    }
}
```

Success response:

```
{
    "jsonrpc": "2.0",
    "id": "pt-1",
    "result": {}
}
```

If the connection ID is already in use an error must be returned. Connection errors (timeouts, etc) are indicated as errors (no specific error codes are in use at present). Connection error descriptions should be as descriptive as possible to help diagnosis.

NetworkDisconnect

Request for an existing connection to be closed. Can also be sent while a NetworkConnect is pending and should abort the connection attempt in a reasonable time (not necessarily immediately).

The terminal may send extra `NetworkDisconnect` requests to ensure the remote state of a connection has been cleaned up. If the connection ID related to such a request is no longer recognized by the ECR, an error should be returned (the terminal will ignore the error).

Request example:

```
{
    "method": "NetworkDisconnect",
    "jsonrpc": "2.0",
    "id": "pt-1",
    "params": {
```

```
        "connection_id": 12,
        "reason": "terminal rebooting"  // optional reason
    }
}
```

Success response:

```
{
    "jsonrpc": "2.0",
    "id": "pt-1",
    "result": {
    }
}
```

Note that ECR must send a `NetworkDisconnected` always when the connection is closed, regardless of whether the disconnection is requested by the remote peer (TCP FIN) or by the terminal using `NetworkDisconnect`.

Data

Plain TCP data to be sent to the Internet for a certain logical connection. The `data` property contains base-64 encoded data. The base-64 format has no spaces or newlines, but does have padding characters (`=`).

Recommended maximum data size per request is 1024 raw bytes which expands to about 1.4kB of JSONPOS data (size limit must not be assumed by receiver). Sending of Data notifys must be rate limited when using RFCOMM to ensure that e.g. `_Keepalive` monitoring won't be disturbed. Fairness across connections must be guaranteed e.g. by sending data for active connections in a round-robin fashion.

Request (notify) example:

```
{
    "method": "Data",
```

```
    "jsonrpc": "2.0",
    "params": {
        "id": 12,
        "data": "<base64 bytes>"
    }
}
```

ECR is required to process `Data` notifys in sequence to ensure they're sent out in the order the terminal sends them.

Screen display with DisplayScreen method

DisplayScreen allows displaying specific screens on terminal. Examples below describe the available generic screens. JSONPOS access to screens must be enabled in the terminal configuration.

Display text ("generic_info")

Displays up to 4 lines of text on terminal screen.

Sample request:

```
{
    "jsonrpc": "2.0",
    "method": "DisplayScreen",
    "id": "pos-1",
    "params": {
        "api_key": "e086554a-9a35-4bbf-9f99-a8a7cd7d1dcc",
        "scr_args": {
            "scr_id": "generic_info",
            "scr_args": {
```

```
                "text1": "A quick brown",
                "text2": "fox jumped",
                "text3": "over lazy dog."
            }
        }
    }
}
```

Info screen is dismissed with JSONPOS Abort method.

Display image ("generic_image")

Displays an image to the screen. The `generic_image` screen includes an image that is invisible by default and should be replaced by an image preloaded to Poplapay backend or submitted by ECR. It is preferable to use images that are smaller than the terminal's screen size.

Sample request (with image preloaded to Poplapay backend):

```
{
    "jsonrpc": "2.0",
    "method": "DisplayScreen",
    "id": "pos-1",
    "params": {
        "api_key": "e086554a-9a35-4bbf-9f99-a8a7cd7d1dcc",
        "scr_args": {
            "scr_id": "generic_image",
            "scr_args": {
                "image_replacements": {
                    "image": "generic_smileys"
                }
            }
```

```
        }
    }
}
```

Sample request (with base64 encoded png image submitted by ECR):

```
{
    "jsonrpc": "2.0",
    "method": "DisplayScreen",
    "id": "pos-1",
    "params": {
        "api_key": "e086554a-9a35-4bbf-9f99-a8a7cd7d1dcc",
        "scr_args": {
            "scr_id": "generic_image",
            "scr_args": {
                "image_replacements": {
                    "image": {
                            "format": "png",
                            "data": "iVBORw0KGgoAAAANSUhEUgAAAMgAAADICAY..."
                    }
                }
            }
        }
    }
}
```

Generic image screen is dismissed with JSONPOS Abort method.

Display a menu ("generic_menu")

Allows user to select from up to 9 menu items.

Sample request:

```
{
    "jsonrpc": "2.0",
    "method": "DisplayScreen",
    "id": "pos-1",
    "params": {
        "api_key": "e086554a-9a35-4bbf-9f99-a8a7cd7d1dcc",
        "scr_args": {
            "scr_id": "generic_menu",
            "scr_args": {
                // Menu title is optional
                "title": "Pick a fruit",
                // Up to 9 menu items can be defined
                "item1_enabled": 1,
                "item2_enabled": 1,
                "item3_enabled": 1,
                "item1_text": "Apple",
                "item2_text": "Orange",
                "item3_text": "Banana"
            }
        }
    }
}
```

Sample response with item2 selected:

```
{
    "jsonrpc": "2.0",
    "id": "pos-1",
    "result": {
        "action": "item2"
    }
}
```

Sample response with user canceling:

```
{
    "jsonrpc": "2.0",
    "id": "pos-1",
    "result": {
        "action": "cancel"
    }
}
```

Display an amount request ("generic_enter_sum")

Sample request:

```
{
    "jsonrpc": "2.0",
    "method": "DisplayScreen",
    "params": {
        "api_key": "e086554a-9a35-4bbf-9f99-a8a7cd7d1dcc",
        "cashier_language": "it",
        "scr_args": {
```

```
            "scr_id": "generic_enter_sum",
            "scr_args": {
                "currency": "EUR"
            }
        }
    }
}
```

Sample response with amount 133 entered:

```
{
    "jsonrpc": "2.0",
    "id": "id-1518602294-2441644",
    "result": {
        "value": "133",
        "action": "accept"
    }
}
```

Sample response with user canceling:

```
{
    "jsonrpc": "2.0",
    "id": "id-1518602276-1779674",
    "result": {
        "value": "",
        "action": "cancel"
    }
}
```

Sample response with user pressing menu:

```
{
    "jsonrpc": "2.0",
    "id": "id-1518602276-1779674",
    "result": {
        "action": "menu"
    }
}
```

Supported currencies vary per device:

- SPm20 supports a full set of currencies (starting from version 18.4.0)
- Other devices support only EUR (up to version 18.3.0) or EUR, GBP, SEK, NOK, DKK (starting from version 18.4.0)

Request clerk number ("generic_clerk_id_entry")

Allows to request clerk number from user.

Sample request:

```
{
    "jsonrpc": "2.0",
    "method": "DisplayScreen",
    "id": "pos-1",
    "params": {
        "api_key": "e086554a-9a35-4bbf-9f99-a8a7cd7d1dcc",
        "scr_args": {
            "scr_id": "generic_clerk_id_entry",
            "scr_args": {}
```

```
        }
    }
}
```

Sample response after user has accepted input:

```
{
    "jsonrpc": "2.0",
    "id": "pos-1",
    "result": {
        "action": "accept",
        "value": "123456"
    }
}
```

Sample response after user has canceled input:

```
{
    "jsonrpc": "2.0",
    "id": "pos-1",
    "result": {
        "action": "cancel",
        "value": ""
    }
}
```

Request user selection ("generic_select")

Allows to request user to select from up to 3 alternatives. Screen includes `image` that is invisible by default and should be replaced by an image preloaded to Poplapay backend or submitted by ECR.

Sample request (with image preloaded to Poplapay backend):

```
{
    "jsonrpc": "2.0",
    "method": "DisplayScreen",
    "id": "pos-1",
    "params": {
        "api_key": "e086554a-9a35-4bbf-9f99-a8a7cd7d1dcc",
        "scr_args": {
            "scr_id": "generic_select",
            "scr_args": {
                "image_replacements": {
                    "image": "generic_select_smileys"
                }
            }
        }
    }
}
```

Sample request (with base64 encoded png image submitted by ECR):

```
{
    "jsonrpc": "2.0",
    "method": "DisplayScreen",
    "id": "pos-1",
    "params": {
        "api_key": "e086554a-9a35-4bbf-9f99-a8a7cd7d1dcc",
```

```
        "scr_args": {
            "scr_id": "generic_select",
            "scr_args": {
                "image_replacements": {
                    "image": {
                            "format": "png",
                            "data": "iVBORw0KGgoAAAANSUhEUgAAAMcAAACwAgM..."
                        }
                    }
                }
            }
        }
    }
}
```

Sample response after user has selected leftmost function key:

```
{
    "jsonrpc": "2.0",
    "id": "pos-1",
    "result": {
        "action": "choice1"
    }
}
```

Request user to input an integer ("generic_3_digit_input")

Allows to request user to input an integer. All terminals accept at lest 3 digits but the number of digits is not necessarily limited to 3 digits. Screen includes a title that can be set in screen parameters.

Sample request:

```
{
    "jsonrpc": "2.0",
    "method": "DisplayScreen",
    "id": "pos-1",
    "params": {
        "api_key": "e086554a-9a35-4bbf-9f99-a8a7cd7d1dcc",
        "scr_args": {
            "scr_id": "generic_3_digit_input",
            "scr_args": {
                "title": "Table number"
            }
        }
    }
}
```

Sample response after user has accepted the input with green key:

```
{
    "jsonrpc": "2.0",
    "id": "pos-1",
    "result": {
        "action": "accept",
        "value": "123"
    }
}
```

Display a prompt for phone number ("generic_phone_number_input")

Sample request:

```
{
    "jsonrpc": "2.0",
    "method": "DisplayScreen",
    "id": "id-1518602294-2441645",
    "params": {
        "api_key": "e086554a-9a35-4bbf-9f99-a8a7cd7d1dcc",
        "cashier_language": "it",
        "scr_args": {
            "scr_id": "generic_phone_number_input",
            "scr_args": {
            }
        }
    }
}
```

Sample response with number 102714230 entered:

```
{
    "jsonrpc": "2.0",
    "id": "id-1518602294-2441645",
    "result": {
        "value": "102714230",
        "action": "accept"
    }
}
```

Sample response with user canceling:

```
{
    "jsonrpc": "2.0",
    "id": "id-1518602294-2441645",
    "result": {
        "value": "",
        "action": "cancel"
    }
}
```

Non-payment-data

Non-payment-data is given out on SwipeEvent, CardInfo and AppInfo method requests. The information given is dependent on the terminal configuration and card used.

Non-payment-data contains a list of data elements, each of which contains information read from the card. These can be e.g. raw magstripe tracks, MIFARE tags, or information related to detected loyalty programs. As an example, raw magstripe data may be given on whitelisted non-payment cards, to enable ECR applications to use these from their own purposes.

The following table illustrates some supported non-payment data examples:

Description	Example
Raw magstripe track 1	`{ "type": "raw", "source": "track1", "track_data": "..." }`

Raw magstripe track 2	<pre>{
	"type": "raw",
	"source": "track2",
	"track_data": "..."
	}</pre>

Raw magstripe track 3	<pre>{
	"type": "raw",
	"source": "track3",
	"track_data": "..."
	}</pre>

MIFARE tag	<pre>{
	"type": "uid",
	"source": "mifare",
	"uid": "804d153a3d6404"
	}</pre>

Loyalty from chip	<pre>{
	"type": "loyalty",
	"loyalty_type": "finnair_plus",
	"source": "chip",
	"aid": "A0000000041010"
	"member_id": "608589925",
	"cardholder_level": "04",
	"partner_id": "00"
	}</pre>

Loyalty from magnetic stripe	<pre>{
	"type": "loyalty",
	"loyalty_type": "finnair_plus",</pre>

	<pre>"source": "track2",
	"member_id": "608589925",
	"cardholder_level": "004",
	"partner_id": "110501"
	}</pre>

Current token for selected application that can be stored	<pre>{
	"type": "token",
	"token_data":
	{
	"value": "PIT1999999999999999999",
	"token_type": "scoped_irreversible",
	"store": true
	}
	}</pre>

Current token for non-selected application that can be stored	<pre>{
	"type": "token",
	"token_data":
	{
	"value": "PIT1999999999999999998",
	"token_type": "scoped_irreversible",
	"store": true
	}
	}</pre>

Obsolete token for selected application that can be used for lookups but must not be stored	<pre>{
	"type": "token",
	"token_data":
	{
	"value": "PIT1999999999999999997",
	"token_type": "scoped_irreversible",
	"store": false</pre>

	<pre>}
	}</pre>

Token received from external tokenization service	<pre>{
	"type": "token",
	"token_data":
	{
	"value": "65a33d610fc69646daca38aaeaa35c5c28d3065ba3aa202e7566a0d4dc756d61",
	"token_type": "external_payment_highway",
	"store": true
	}
	}</pre>

Indication that there has been an error in tokenization causing incomplete tokenization results	<pre>{
	"type": "tokenization_error",
	"error":
	{
	"code": "INTERNAL_ERROR",
	"description": "Error: INTERNAL_ERROR: Token...",
	"details": "Error: INTERNAL_ERROR: Token..."
	}
	}</pre>

Device specific limitations:

- SPm20 does not support reading of track 3 (hardware limitation).

Network proxy

Overview

Some terminal types don't have TCP/IP connectivity of their own and may only have (for example) an RFCOMM serial interface JSONPOS allows such terminals to get Internet access via the JSONPOS network proxy feature. The related methods are described above, with basic method usage as follows:

- If RFCOMM or USB serial is used, the ECR uses `_Sync` to bring the JSONPOS connection into synchronized state.
- The ECR sends `NetworkStart` to enable connection proxying.
- When the terminal needs a new connection, it sends out `NetworkConnect`.
- If the connection is successful, the terminal and the ECR exchange connection related data segments using overhead minimized `Data` notifications. Data transfer must be rated limited.
- The terminal may request the connection to be closed using `NetworkDisconnect`. The connection may also be closed by the remote Internet peer.
- When a disconnect is pending, the ECR delivers all pending data to the terminal and then sends a `NetworkDisconnected` request to indicate the connection is finished from the ECR point of view.

Each connection is identified using a numeric connection ID. Multiple connections may exist in various states simultaneously. If the JSONPOS connection is lost (which includes a `_Sync` based resynchronization) all network connections related to the lost connection should be closed by both ends automatically.

See the individual method descriptions for details.

Rate limiting

Because RFCOMM links have a low throughput, data transfers must be rate limited to ensure robust operation of the JSONPOS connection. If rate limiting is not done, it's possible for keepalive requests fail due to data messages consuming all RFCOMM bandwidth, which affects terminal reliability.

Two rate limit algorithms are recommended:

- Impose a write rate limit for the RFCOMM serial stream as a whole, for example 10kB/sec. This limit is device specific and necessary to ensure too much RFCOMM data is not queued up which may cause a connection to fail or to react too slowly to e.g. keepalives.

- Impose a write rate limit for Data messages, ensuring that Data messages (for all proxied connections combined) in their encoded form use significantly less bandwidth than the RFCOMM write rate limit. For example, assuming a 1.4x expansion, which roughly accounts for JSON-RPC framing and base-64 encoding of data, a good Data message limit might be 5kB/s of raw data. That would expand to about 7kB/sec of JSON-RPC data, leaving 3kB/sec free for other JSONPOS requests.

Each sending peer must ensure fairness across active proxied connections so that data for each connection is sent even when one or more connections have a large transfer backlog. The easiest approach to ensure this is to use a round-robin algorithm and send rate limited data for each active connection in turn.

Without fairness guarantees it's possible for a large file download to starve the PSP server connection (which uses keepalives) causing a PSP connection drop.

Other notes

- There's currently no specific support for half-open TCP connections.

Bluetooth RFCOMM

JSONPOS can be used over Bluetooth RFCOMM. Main differences to TCP:

- Bluetooth pairing, service discovery, and RFCOMM channels are device specific.
- The ECR uses a `_Sync` transport-level request to initialize and resynchronize a JSON-RPC transport connection.
- Connections delimited by `_Sync` are considered separate logical JSON-RPC connections but share the same RFCOMM link.
- The terminal sends frequent `_Keepalive` requests to the ECR when using Bluetooth. Responding to `_Keepalive` is mandatory.

The network proxy methods are independent of Bluetooth; they can also be used with TCP, and RFCOMM can be used with and without network proxy methods.

USB serial

JSONPOS can be used over USB serial. Behavior is similar to RFCOMM, i.e. a `_Sync` handshake is required to (re)synchronize the connection state.

Device information

Feature	VEGA3000 (V3M2)	VEGA3000 (V3P3)	YOMANI	YOXIMO	VALINA	SPm20	Notes
Chip reader	x	x	x	x	x	x	
NFC reader	x	x	x	x	x	x	
Mag track 1	x	x	x	x	x	x	
Mag track 2	x	x	x	x	x	x	
Mag track 3			x	x	x		SPm20 hardware does not support track 3 reading.
Ethernet JSONPOS		x	x		x		
USB Ethernet JSONPOS			x				
Wi-Fi JSONPOS	x			x			
3G JSONPOS	x		x	x			
Bluetooth RFCOMM JSONPOS						x	Network connectivity using JSONPOS network proxy. SPm20 RFCOMM channel 1, SPP UID 00001101-0000-1000-8000-00805F9B34FB, iAP

							external accessory protocol name to be added to Info.plist of an iOS app: com.thyron.
USB serial JSONPOS						x	Network connectivity using JSONPOS network proxy.

Payment terminal IP address discovery options

When using LAN connection to PT, ECR systems need to know the IP address of the payment terminal to be able to interface with the terminal. For getting the IP address for ECR system there are several options:

1. Configure LAN router to provide static IP address for the terminal. Utilize MAC address from payment terminal label and configure DHCP server to provide static IP address for the specific MAC address. Then use this IP address in ECR system.
2. Use Link Local Multicast Name resolution (LLMNR). Yomani and Yoximo terminals support local IP address discovery using name in format `poplatek-<MAC address>` or `<terminal name>` where `<terminal name>` is a name generated from payment terminal name shown in PoplaView service.
3. Use Multicast DNS (mDNS). mDNS names for payment terminals are LLMNR names extended with `.local` suffix.
4. For terminal connected with USB cable to Windows PC, use name samoa.mshome.net.

Option 1 is applicable for Yomani, Yoximo and Valina terminals. Options 2 and 3 are applicable to Yomani and Yoximo terminals. Option 4 is applicable to Yomani terminals. LLMNR is supported in Windows Vista and later. mDNS is supported in most Linux distributions and Apple computers.

Deprecated features

- The following common fields in ECR messages are not used by the terminal and were removed from the specification: `timestamp`, `pos_id`, `sale_location_code`, `cashier_code`, `protocol_version`. ECR specific identifiers can be associated with a transaction using the `external_data` field.

- The following common fields in PT messages were removed from the specification: `timestamp`, `pt_hardware_id`, `pt_logical_id`, `pt_version`, `pt_revision`, `protocol_version`.
- The `cashier_language` field send by ECR is only processed by specific methods (e.g. Purchase, Refund, Cancel, DisplayScreen). The field does not need to be included in any other messages.
- The VersionInfo method has been renamed to TerminalInfo.
- Status response `details` field has been deprecated. `details.pt_name` has been moved to TerminalInfo response `name` field. `details.sales_location_name` have been moved to TerminalInfo response field of the same name.
- DisplayScreen `generic_enter_sum_eur` has been deprecated, use `generic_enter_sum` with `currency` argument set to `EUR` instead.
- Purchase request parameters `stop_on_loyalty and stop_on_app_info` have been deprecated. Use `request_app_info` instead.
- In CardInfo `card_applications` item fields `lookup_tokens`, `store_token`, `tokenization_error_code`, `tokenization_error_description` and `tokenization_error_details` have been deprecated, use `non_payment_data` instead.
- AppInfo fields `store_token`, `lookup_tokens`, `tokenization_error_code`, `tokenization_error_description` and `tokenization_error_details` have been deprecated, use `non_payment_data` instead.
- TerminalInfo printer `max_heigth` field (with a typo) is deprecated and has been removed in terminal version 20.2. Use `max_height` instead.
- Purchase field `bypass_pin` is removed after 21.0.7 release due to MasterCard requirement.
- Support for ECR provided transaction reference number and timestamp have been deprecated and removed in terminal version 21.2.
- DisplayScreen `generic_menu` response field `value` has been deprecated.
- The informative `response_to` field in method responses has been deprecated. The field is non-programmatic, i.e. it must not be processed programmatically.