JSONPOS Integration Guide

Introduction

This document summarizes how to implement an electronic cash register (ECR) - payment terminal (PT) integration using JSONPOS to maximize robustness and automatic recovery in error situations. For details of the specific JSONPOS requests, see the JSONPOS protocol specification.

Integration checklist

Transport handling

- Check that `_CloseReason` is sent before closing a connection when possible.
- Check that ECR responds to `_Keepalive` requests in all situations (including pending Purchase or other request).
- Check that ECR monitors the connection using `_Keepalive` reliably, see discussion below. Keepalive monitoring is the most important connection reliability mechanism because it covers all stack layers in one mechanism.

- Check that ECR correctly parses fragmented JSON-RPC frames. An easy torture test is to simulate a situation where each transport read call returns only one byte of data. Everything must work correctly even in this case.
- Check that ECR correctly parses multiple JSON-RPC frames from one transport layer read() call.
- Check that ECR JSONRPC parser deals reliably with a frame that never completes with a sanity timeout. For example, if one or more frame bytes are received and the frame doesn't complete within 10 seconds, the transport should be closed. Keepalive monitoring also serves this purpose: if a frame cannot be completed, also keepalive request/reply initiated by ECR would usually fail and cause transport close.
- Check that ECR correctly sends and receives non-ASCII Unicode characters.
- Check that ECR **always** responds to inbound requests at some point. Terminal assumes that if keepalives are functional, every request will be eventually processed to completion (success or error). For example, if the terminal sends a NetworkConnect request, ECR must always respond with success/error or error eventually (even in connection timeouts etc).
- Check that ECR doesn't require or interpret `response_to` field which is for human readability only and deprecated.

Transaction reliability

- See transaction reliability notes below.

Network proxy (if implemented by ECR)

- Trigger three parallel Test methods with `test_id: "large_file_download"` and ensure that the downloads complete without the JSONPOS connection being dropped due to e.g. timeout, and that normal purchase use cases work reasonably well even when downloads are in progress. This exercises network proxy rate limiting and fairness.
- When remote peer closes a TCP connection, ensure that any pending ECR-to-terminal Data notifies are sent to completion before sending a NetworkDisconnected to the terminal.

JSONPOS Test method

The JSONPOS protocol has a `Test` method intended for development time stress testing of the ECR implementation. A test can be initiated by sending a test request:

```
{
    "jsonrpc": "2.0",
    "method": "Test",
    "id": "xxx",  // replace with dynamic ID
    "params": {
        "api_key": "xxx", // replace with actual API key
        "test_id": "xxx"  // replace with actual test ID
        // possible additional test parameters
    }
}
```

The available test IDs may change arbitrarily between terminal software releases. Current tests include:

test_id	Description
large_file_download	Download a ~1MB file and verify it downloads correctly.

Transport connection monitoring

JSON-RPC `_Keepalive` is the primary mechanism

The JSON-RPC transport connection should be primarily monitored using periodic JSON-RPC `_Keepalive` requests initiated by the ECR. This is preferable over lower level mechanisms like TCP/IP pinging or Bluetooth RFCOMM monitoring because it ensures end-to-end functionality of the whole connection stack with a single mechanism.

The `_Keepalive` mechanism allows each peer to monitor the other peer independently. Each peer can freely select the interval when keepalive requests are sent; the other peer is only required to respond to incoming keepalive requests promptly (even when processing some longer term request like a network connection or a purchase). The ECR is thus free to e.g.:

- Use a longer keepalive interval in idle mode and a shorter one when a purchase or operation is active, or about to start.
- Send additional keepalive requests to ensure connection functionality just before some critical points in code. For example, ECR may send a keepalive request just before starting a transaction.

Other monitoring mechanisms

The ECR should use other mechanisms for connection status monitoring where available, as such mechanisms may be faster than `_Keepalive` based monitoring. For example:

- If a TCP connection is closed, the transport is certainly lost and recovery can begin immediately. There's no reason to wait for `_Keepalive` timeout.
- If RFCOMM is used and the Bluetooth stack provides status information indicating that the RFCOMM connection was lost, recovery can similarly begin immediately.

Recovering from transport connection drop

Pending requests must be terminated with error

When a transport connection drop is detected, the ECR should first ensure all pending requests which were initiated over the lost connection are terminated. Conceptually each such request should fail with an error, just as if the terminal had sent an error reply. If the terminal sent a `_CloseReason`, it may be associated with the terminated pending requests as a possible cause.

It's important for nothing to remain pending if a connection is lost, as there will never be a reply from the terminal to the pending requests, even after establishing a new connection (requests are bound to a specific connection).

Reinitializing the connection

Once pending requests have been dealt with, the ECR should reconnect to the terminal. The specific mechanism depends on the transport in question (TCP/IP or RFCOMM).

For RFCOMM, it is important to:

- Close current RFCOMM connection (if not already closed) and reconnect to the terminal.
- Before sending a `_Sync`, read and ignore data from the RFCOMM connection for several seconds; 5 seconds is a good starting point. Sometimes data belonging to an earlier connection will trickle out a reconnected RFCOMM connection because RFCOMM can in many cases resume a previous connection.
- Send a `_Sync` and wait for a response for several seconds. It's important to use a unique request ID for each `_Sync`, and ensure that the response received matches the request ID. This ensures that the sync response is not an old one. If no sync response is received in a reasonable time, say 5 seconds, fail the connection attempt and retry.
- If the process fails at any point, disconnect RFCOMM and retry from start. It's a good idea to reconnect the RFCOMM as far as the Bluetooth stack is concerned, so that any hanging connection situations are resolved as reliably as possible.

In rare situations it may be that some operating system or application state in the ECR or the terminal prevents a transport connection from forming. For example, maybe a Bluetooth driver is stuck and fails to process any data despite repeated connection attempts. The ECR can recover from this situation by:

- Keeping track of the number of failed consecutive connection attempts.
- If the count is high enough, reboot the ECR and then resume reconnecting.
- At present there's no automatic reboot in the terminal from lack of Bluetooth connectivity, because there's no expectation that a Bluetooth connection is constantly maintained. The ECR UI might suggest for the user to reboot the terminal manually.

RFCOMM pairing

The RFCOMM Bluetooth pairing process is quite complicated and may sometimes fail. It should be easy to retry pairing from the ECR UI.

Once paired, there are no known issues for pairing state to be lost.

JSON-RPC parsing reliability

JSON-RPC message fragmentation

When parsing JSON-RPC framed message it's critical to avoid any assumptions about TCP/RFCOMM transport read calls returning complete messages or even complete message fields. In particular:

- When reading the length prefix (8 hex digits) of a JSON-RPC message, there are no guarantees that it arrives in one read call.
- When doing a read(), the ECR may receive a partial message, but may also receive multiple complete messages in addition to a possible partial one. All of the completes messages must be processed before waiting for new data to arrive.

These are important to be correctly implemented because the corner cases like partial length field or multiple messages arriving in one read() are rare. They do happen, though, and if handled incorrectly, this leads to very difficult-to-diagnose problems. What's worse, an update in the terminal software may affect how JSON-RPC messages are fragmented across TCP/RFCOMM reads which may then trigger a production issue in the ECR integration.

One good "torture test" is to use a debug build where the ECR reads incoming JSON-RPC data one byte at a time. The JSON-RPC message parsing and processing should work flawlessly even in this case.

JSON issues

The messages are in JSON format which is well documented. If a custom JSON parser is used, it's important to stress test it. Particular issues that should be tested:

- Handling of non-ASCII string data. Most messages are pure ASCII so failure to handle non-ASCII correctly may lead to odd production issues.
- At present non-ASCII characters are escaped in the JSON-RPC layer so that the resulting encoded message is ASCII only. This may change in the future, and implementing UTF-8 parsing is recommended.

Transaction reliability

Pending requests and Purchase/Check

When the JSON-RPC connection is lost, the ECR has no way of knowing what happened to pending requests:

- The terminal may not have received the request at all.
- The terminal may have received the request, but failed to process it because it detected that the transport connection was lost.
- The terminal may have received the request, processed it, and sent a reply, but the reply was lost before it received the ECR.

For Purchases it is critical to ensure that the ECR can reliably figure out what happened to any initiated Purchase. To support this goal, JSONPOS provides a `Check` method which allows the ECR to check what happened to a previously initiated Purchase. The Check request may be sent as many times as needed.

The Check response contains exactly the same data as the corresponding Purchase response, even across terminal reboots (the data is persisted to flash). The Check request can thus be used to complete any pending purchase reliably regardless of connection drops. Note that Check may need to be attempted multiple times in case the transaction is still on-going.

Reliable ECR transaction processing

The best possible approach for ECR is to recover from both terminal and ECR reboots as follows:

- When the ECR is about to start a Purchase, allocate IDs for the Purchase and write them to ECR persistent storage.
- Initiate the Purchase with the persisted ID information.
- If Purchase response is received, process the result and update the state in persistent storage to indicate the transaction is finished.
- If the JSON-RPC connection is lost, reconnect, and use `Check` request to complete the purchase. If `Check` result indicates transaction is still active, periodically resend `Check` until the transaction is over.
- If the ECR reboots, use the persisted state to detect that a Purchase is in progress and use `Check` request to figure out what happened. If the ECR rebooted before the initial Purchase request reached the terminal, the terminal will indicate "transaction unknown" which is a reliable indication that no purchase was processed at all.

This processing should be manually tested for each connection drop, terminal reboot, ECR reboot condition. As far as the JSONPOS protocol is concerned, it should be possible to process the transaction to completion reliably in every situation.

Network proxying

Network proxy support, i.e. NetworkStart, NetworkConnect, NetworkDisconnect, is used with Bluetooth RFCOMM. It's important for the proxying to be robust, because any bugs in proxying code lead to very difficult-to-diagnose problems that may come and go over time.

There's no specific technique to write robust proxying code, but general recommendations include:

- Stress test parallel connections carefully. The terminal establishes multiple parallel connections in normal operation.
- Ensure that the ECR handles connection closure correctly. Specifically, the ECR must deliver all pending data before sending a NetworkDisconnected. Otherwise the terminal will fail to receive some of the pending data which can lead to terminal communication failures.
- Ensure that ECR rate limiting mechanisms are robust. It's important that data traffic doesn't saturate the RFCOMM link; otherwise keepalive monitoring may fail which brings down all network connections.
- Ensure that network connection state is terminated and any connections are closed if the JSON-RPC transport is closed. Any open TCP connections cannot be continue across JSON-RPC transport reconnection, so it's critical to always close the connections reliably to avoid e.g. leaking connection state.