Application Note

READER HOW-TO

HEALTH MONITORING

Version 1.00

www.impinj.com

This document provides a description of how to monitor the health of an Impinj Speeday® RAIN RFID reader.

TABLE OF CONTENTS

INTRODUCTION

The following document describes how to use the built-in features of Impinj Speedway® RAIN RFID readers to detect various failures, including those related to network, antenna connectivity issues and internal software. Hardware failures of the Speedway readers are rare, but can still affect the performance of an RFID system. When a hardware failure occurs, Speedway can use Low Level Reader Protocol (LLRP) communications to recover automatically, if LLRP communications are configured to detect failures and re-establish connectivity.

This document covers implementation details and applicable .NET C# code examples to show how to configure a failure detection feature with the Impinj Octane™ Software Development Kit (SDK). The examples used are based on the 2.32.1 release of Octane SDK.

KEEP ALIVES (READER-CLIENT HEARTBEAT)

Speedway readers support the LLRP Keep Alive mechanism. LLRP Keep Alive provides a 'heartbeat' communication between the reader and client applications via `KEEPALIVE` and `KEEPALIVE_ACK` messages. The reader sends periodic `KEEPALIVE` messages to the client application, which responds with a `KEEPALIVE_ACK` acknowledgement message. If the reader fails to receive a determined number of consecutive `KEEPALIVE_ACK` messages, it will close the current LLRP connection and make itself available for subsequent LLRP connections. The user can reconfigure the `KEEPALIVE` timeout number if desired. Similarly, if the client application stops receiving the periodic `KEEPALIVE` messages from the reader, it registers a likely network connection issue and can close the current LLRP socket so it can re-establish connection with the reader. With this feature, both the client application and the reader can detect a potential connection issue and respond in a graceful manner.

In the code sample below, based on the `Keepalives` example C# project in the Octane SDK .NET distribution, implementing the Keep Alive feature takes two steps. In the first step, the user enables the feature and defines how frequently the reader will send `KEEPALIVE` messages to the client application. In the second step, the user enables the reader `LinkMonitor` mode that tells the reader to reset the LLRP connection if it fails to receive the `LinkDownThreshold` number of `KEEPALIVE_ACK` acknowledgement messages from the client application. The Octane SDK automatically sends the `KEEPALIVE_ACK` message from the client application back to the reader in response to any `KEEPALIVE` messages.

> **NOTE:** The user can also configure the Keep Alive feature without enabling `LinkMonitor`; in this setup, the reader still sends periodic messages to the client application, but doesn't track any `KEEPALIVE_ACK` messages sent in reply.

Once the Keep Alive mechanism is enabled and configured, the user must define how the client application determines a lost reader connection by using the `ConnectionLost` event. If desired, the user can also configure notifications to alert the client application each time that a `KEEPALIVE` message is received by using the `KeepaliveReceived` event. Event handlers are bound to each of these events in the code below:

```
reader.Connect(hostname);

// Get the default settings
// We'll use these as a starting point
// and then modify the settings we're
// interested in.
Settings settings = reader.QueryDefaultSettings();

// Enable keepalives on the reader. Once enabled,
// the reader will send a KEEPALIVE message to
// the client application every PeriodInMs milliseconds
settings.Keepalives.Enabled = true;
settings.Keepalives.PeriodInMs = 3000;

// Enable link monitor mode.
// If our application fails to reply to
// five consecutive keepalive messages,
// the reader will close the network connection.
settings.Keepalives.EnableLinkMonitorMode = true;
settings.Keepalives.LinkDownThreshold = 5;

// Assign an event handler that will be called
// if the reader stops sending keepalives.
reader.ConnectionLost += OnConnectionLost;

// (Optional) Assign an event handler that will
// be called when keepalive messages are received.
reader.KeepaliveReceived += OnKeepaliveReceived;

// Apply the newly modified settings.
reader.ApplySettings(settings);
```

Configuring and using KeepAlives

Using the Keep Alive mechanism, both the reader and client application can keep track of network connectivity and respond to any disruptions.

Minimizing the impact of network connectivity on tag reads

When network disruptions occur, Speedway won't report any tag reads that occur between the disconnect and reconnect events by default. The user can configure the reader to retain tag reads until it re-establishes a connection, by enabling the "Hold Reports on Disconnect" setting:

```
// Tell the reader to hold all tag reports and events
// when we disconnect from the reader.

settings.HoldReportsOnDisconnect = true;
```

The user can then prompt the reader to resume sending events and reports after it re-establishes a connection to the client application with the "Resume Events and Reports" event shown below:

```
// Reconnect to the reader.
reader.Connect(hostname);

// Enable tag reports and events.

reader.ResumeEventsAndReports();
```

Note that the user must perform these operations in a separate thread from that of the `ConnectionLost` event handler.

The `DisconnectedOperation` project in the Octane SDK examples presents an example of how to do this.

READER EVENTS

Speedway has a built-in reader event mechanism which can detect an antenna disconnection. The user can configure the reader to report this event to alert the client application. Reader events can also detect the following:

- Change of state of the General-Purpose Inputs (GPIs)
- Reader started events
- Reader stopped events

All the reader events are configured by binding an event handler to each of the available reader events, per the following code example:

```
reader.Connect(hostname);

// Get the default settings
// We'll use these as a starting point
// and then modify the settings we're
// interested in.
Settings settings = reader.QueryDefaultSettings();

// Enable all of the antenna ports.
settings.Antennas.EnableAll();

 // Apply the newly modified settings.
 reader.ApplySettings(settings);

 // Assign handlers for various reader events.
 reader.GpiChanged += OnGpiEvent;
 reader.AntennaChanged += OnAntennaEvent;
 reader.ReaderStarted += OnReaderStarted;
 reader.ReaderStopped += OnReaderStopped;

 // Start the reader (required for antenna events).
 reader.Start();
```

Configuring and using Reader Events

With the `AntennaChanged` events, the reader will report when any antenna has changed state from `Disconnected` to `Connected`, and vice versa. Note that the `AntennaChanged` detection feature only works when the reader is performing an inventory.

SNMP TRAPS

Speedway's built-in SNMP daemon can help diagnose the cause of a reader disconnection. If the reader reboots, the SNMP daemon will issue SNMP trap notifications for the following events:

Notification	SNMP Details
Reader power up	• `coldStart` notification defined in the SNMPv2-MIB Management Information Base (MIB) • OID .1.3.6.1.6.3.1.1.5.1
Reader shut down	• `nsNotifyShutdown` notification defined in NET-SNMP-AGENT-MIB • OID .1.3.6.1.4.1.8072.4.0.2
Reader restart	• `nsNotifyRestart` notification defined in NET-SNMP-AGENT-MIB • OID .1.3.6.1.4.1.8072.4.0.3
Reader restarts in response to an error condition	• `impUnexpectedRestart` notification defined in IMPINJ-ROOT-REG-MIB • OID .1.3.6.1.4.1.25882.4.1

These trap notifications will inform the user whether a lost reader connection was due to an unexpected reader reboot, or some other cause, such as a network connectivity issue. In order to use these traps, the user must enable and configure the SNMP on the reader and define a destination for the SNMP trap events, using the Speedway console interface, or RShell. Once configured, the reader will issue an SNMP trap notification in response to each event.

The user can configure the Speedway SNMP using the following RShell commands:

Purpose	Rshell Command	Example
Enable SNMP	`config snmp service enable`	
Enable SNMP Trap Service	`config snmp trapservice enable`	
Enable Unexpected restart trap event	`config snmp trap enable unexpectedrestart`	
Configure destination for trap events	`config snmp trap sink <hostname>`	`config snmp trap sink speedway-12-13-14`

Detecting SNMP Traps in Code

The following code sample, based on the SnmpSharpNet open source C# library available at https://www.snmpsharpnet.com, demonstrates how to capture the SNMP traps from a reader programmatically so that the client application can respond to any unexpected error conditions:

```
using System;
using System.Net;
using System.Net.Sockets;
using SnmpSharpNet;
namespace traprecv
{
    class Program
    {
        const string COLD_START_NOTIFICATION_OID = @"1.3.6.1.6.3.1.1.5.1";
        const string SHUTDOWN_NOTIFICATION = @"1.3.6.1.4.1.8072.4.0.2";
        const string RESTART_NOTIFICATION = @"1.3.6.1.4.1.8072.4.0.3";
        const string AUTHENTICATION_FAILURE_NOTIFICATION = @"1.3.6.1.6.3.1.1.5.5";
        const string IMPINJ_UNEXPECTED_RESTART_NOTIFICATION = @"1.3.6.1.4.1.25882.4.1";

        static void Main(string[] args)
        {
            // Construct a socket and bind it to the trap manager port 162
            Socket socket =
                new Socket(
                    AddressFamily.InterNetwork,
                    SocketType.Dgram,
                    ProtocolType.Udp
                    );
            IPEndPoint ipep =
                new IPEndPoint(IPAddress.Any, 162);
            EndPoint ep = (EndPoint)ipep;
            socket.Bind(ep);
            // Disable timeout processing. Just block until packet is received
            socket.SetSocketOption(
                SocketOptionLevel.Socket,
                SocketOptionName.ReceiveTimeout,
                0
                );
            bool run = true;
            int inlen = -1;
            while (run)
            {
                byte[] indata = new byte[16 * 1024];
                // 16KB receive buffer int inlen = 0;
                IPEndPoint peer = new IPEndPoint(IPAddress.Any, 0);
                EndPoint inep = (EndPoint)peer;
                try
                {
                    inlen = socket.ReceiveFrom(indata, ref inep);
```

```
                }
                catch (Exception ex)
                {
                    Console.WriteLine("Exception {0}", ex.Message);
                    inlen = -1;
                }
                if (inlen > 0)
                {
                    // Check protocol version int
                    int ver = SnmpPacket.GetProtocolVersion(indata, inlen);
                    if (ver == (int)SnmpVersion.Ver1)
                    {
                        // Parse SNMP Version 1 TRAP packet
                        SnmpV1TrapPacket pkt = new SnmpV1TrapPacket();
                        pkt.decode(indata, inlen);
                        Console.WriteLine("** SNMP Version 1 TRAP received from {0}:",
                            inep.ToString());
                        Console.WriteLine("*** Trap generic: {0}", pkt.Pdu.Generic);
                        Console.WriteLine("*** Trap specific: {0}", pkt.Pdu.Specific);
                        Console.WriteLine("*** Agent address: {0}",
                            pkt.Pdu.AgentAddress.ToString());
                        Console.WriteLine("*** Timestamp: {0}", pkt.Pdu.TimeStamp.ToString());
                        Console.WriteLine("*** VarBind count: {0}", pkt.Pdu.VbList.Count);
                        Console.WriteLine("*** VarBind content:");
                        foreach (Vb v in pkt.Pdu.VbList)
                        {
                            Console.WriteLine(
                                "**** {0} {1}: {2}",
                                v.Oid.ToString(),
                                SnmpConstants.GetTypeName(v.Value.Type),
                                v.Value.ToString()
                                );
                        }
                        Console.WriteLine("** End of SNMP Version 1 TRAP data.");
                    }
                    else
                    {
                        // Parse SNMP Version 2 TRAP packet
                        SnmpV2Packet pkt = new SnmpV2Packet();
                        pkt.decode(indata, inlen);
                        Console.WriteLine(
                            "** SNMP Version 2 TRAP received from {0}:",
                            inep.ToString()
                            );
                        if ((SnmpSharpNet.PduType)pkt.Pdu.Type != PduType.V2Trap)
                        {
                            Console.WriteLine("*** NOT an SNMPv2 trap ****");
                        }
                        else
                        {
                            string trapObjectId = pkt.Pdu.TrapObjectID.ToString();

                            switch(trapObjectId)
                            {
                                case COLD_START_NOTIFICATION_OID:
                                    Console.WriteLine("*** Cold Start Notification Trap
received");
                                    break;
                                case SHUTDOWN_NOTIFICATION:
                                    Console.WriteLine(
                                        "*** Shutdown Notification Trap received"
                                        );
                                    break;
                                case RESTART_NOTIFICATION:
                                    Console.WriteLine(
                                        "*** Restart Notification Trap received"
                                        );
                                    break;
                                case AUTHENTICATION_FAILURE_NOTIFICATION:
                                    Console.WriteLine(
```

```
                                        "*** Authentication Failure Notification Trap received"
                                        );
                                    break;
                                case IMPINJ_UNEXPECTED_RESTART_NOTIFICATION:
                                    Console.WriteLine(
                                        "*** Unexpected Restart Notification Trap received"
                                        );
                                    break;
                                default:
                                    Console.WriteLine(
                                        "*** Community: {0}",
                                        pkt.Community.ToString()
                                        );
                                    Console.WriteLine(
                                        "*** VarBind count: {0}",
                                        pkt.Pdu.VbList.Count
                                        );
                                    Console.WriteLine(
                                        "*** VarBind content:"
                                        );
                                    Console.WriteLine(
                                        "**** Pkt OID: {0}",
                                        pkt.Pdu.TrapObjectID
                                        );
                                    foreach (Vb v in pkt.Pdu.VbList)
                                    {
                                        Console.WriteLine("**** {0} {1}: {2}",
                                           v.Oid.ToString(),
                                           SnmpConstants.GetTypeName(v.Value.Type),
                                           v.Value.ToString()
                                           );
                                    }
                                    Console.WriteLine("** End of SNMP Version 2 TRAP data.");
                                    Console.WriteLine();
                                    break;
                            }
                        }
                    }
                }
                else
                {
                    if (inlen == 0)
                        Console.WriteLine("Zero length packet received.");
                }
            }
        }
    }
}
```

NOTICES

Impinj gives no representation or warranty, express or implied, for accuracy or reliability of information in this document. Impinj reserves the right to change its products and services and this information at any time without notice.

EXCEPT AS PROVIDED IN IMPINJ'S TERMS AND CONDITIONS OF SALE (OR AS OTHERWISE AGREED IN A VALID WRITTEN INDIVIDUAL AGREEMENTWITH IMPINJ), IMPINJ ASSUMES NO LIABILITY WHATSOEVER AND IMPINJ DISCLAIMS ANY EXPRESS OR IMPLIEDWARRANTY, RELATED TO SALE AND/OR USE OF IMPINJ PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT.

NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY PATENT, COPYRIGHT, MASKWORK RIGHT, OR OTHER INTELLECTUALPROPERTY RIGHT IS GRANTED BY THIS DOCUMENT.

Impinj assumes no liability for applications assistance or customer product design. Customers should provide adequate design and operating safeguards to minimize risks.

Impinj products are not designed, warranted or authorized for use in any product or application where a malfunction may reasonably be expected to cause personal injury or death, or property or environmental damage (“hazardous uses”), including but not limited to military applications; life-support systems; aircraft control, navigation or communication; air-traffic management; or in the design, construction, operation, or maintenance of a nuclear facility. Customers must indemnify Impinj against any damages arising out of the use of Impinj products in any hazardous uses

The products referenced in this document may be covered by one or more U.S. patents. See www.impinj.com/patents for details.