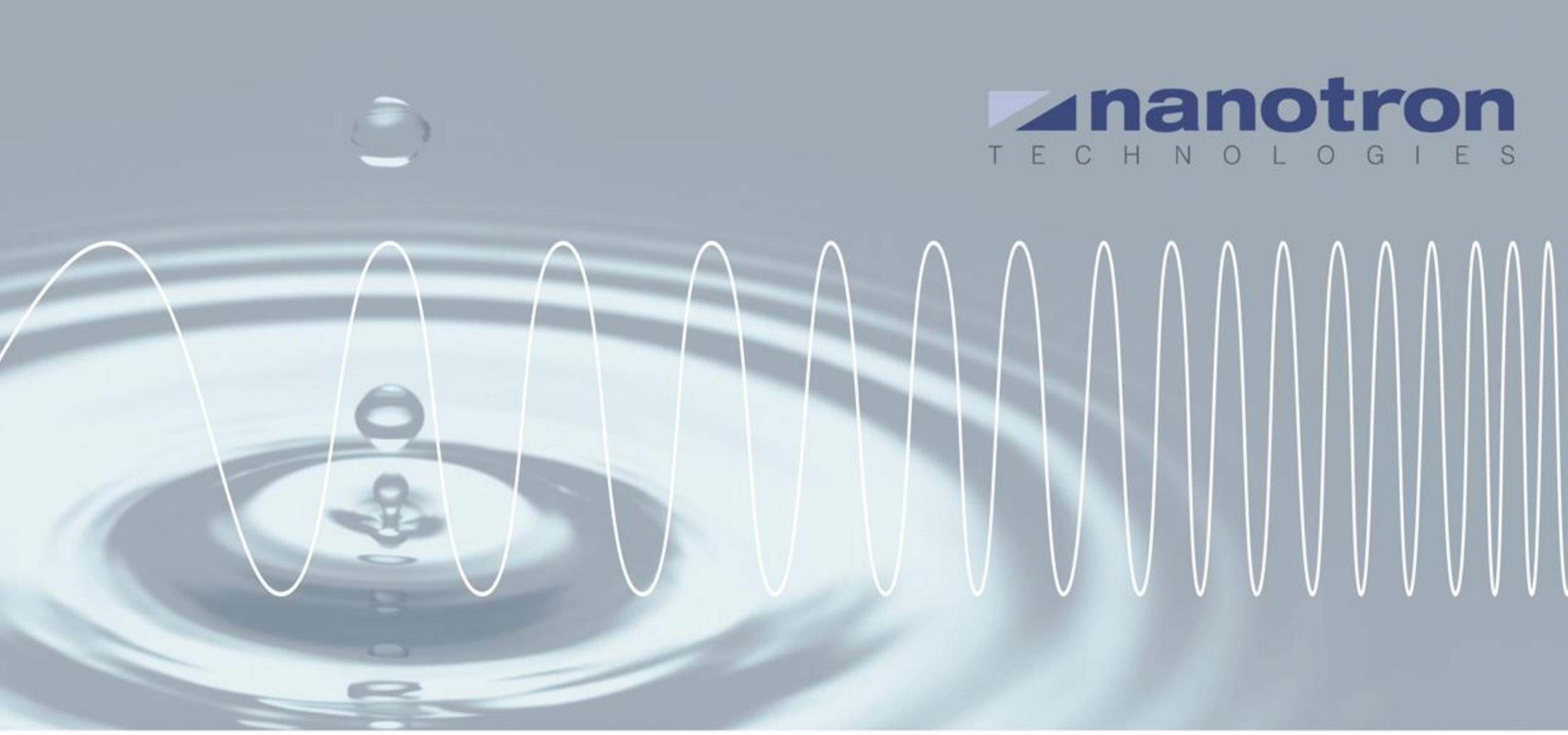

AN0601 Using the binary interfaces of nanoLES

1.2

NA-15-0243-0020-1.2

Document Information

Document Title:	AN0601 Using the binary interfaces of nanoLES
Document Version:	1.2
Current Date:	2016-09-13
Print Date:	2016-09-13
Document ID:	NA-15-0243-0020-1.2
Document Author:	MLA

Contents

1. Introduction 4
   1.1. Purpose 4
2. Google Protocol Buffers 4
   2.1. Format specification 4
3. Getting started 5
   3.1. C++ 5
   3.2. Python 5
   3.3. Java 5
4. How to access the data 6
   4.1. Example using Python 6
   4.2. Example using C++ 7
5. Available binary interfaces 9
      RSSI interface 9
      Result interface 9

Figure 2-1 Format specification of the RSSI port interface 5
Figure 4-1 Format specification of Ressult interface 6
Figure 4-2 Python code in the application client to interpret data at the Result port 7
Figure 4-3 C++ code in the application client to interpret data at the RSSI port 8

1. Introduction

Real Time Location Systems (RTLS) are used in a very broad range of applications. In order to address this diversity the Location Engine Server of nanotron's *find* product line calculates tag location estimates in a generic way that is suitable for many different application fields. As a consequence nanoLES cannot incorporate application specific knowledge into the estimates. Thus users are encouraged to apply application specific knowledge in a post processing phase to achieve optimal results. Often this requires additional information to the location estimates provided at the standard text based user interface of nanoLES. The structure and content of this information varies depending on the selected interface; the format of all these interfaces is binary in order to make the amount of data provided controllable. To ease the access to it they all have been implemented following Google Protocol Buffers.

1.1. Purpose

This document describes the format of the binary data interface of nanoLES. The reader shall be enabled to implement an application that can read the binary data format provided at this interface such that it can be used for further processing. It is assumed that the reader is familiar with software development and common terms used in this field. This document solely describes the format of the binary data and how it can be represented in a data structure. It is not meant as a tutorial for reading data streams since that depends largely on the platform and programming language that is used. Readers unfamiliar with this are advised to consult the documentation of the corresponding programming language.

2. Google Protocol Buffers

The binary interface uses Google Protocol Buffers to specify the data format. This eases the development of client software because the necessary data structures with convenient access methods can be automatically generated from the protocol description provided to the client developer. At the time of writing this document protocol buffers offers support for C++, Python and Java, giving great flexibility to the client developer. A more detailed introduction to Google protocol buffers can be found under the link https://developers.google.com/protocol-buffers/docs/overview.

2.1. Format specification

The format of the binary interface consist of a header and an information set. The header is used to keep track of version information and to indicate the start of the binary stream. The data contained in the information set varies from one specification file to another; but they all have the same structure. It typically consist of the usual blink information provided by the user interface and additional entries for each anchor that received said blink.
Feeding the specification file (provided by Nanotron) to the protocol buffers code generator will generate a set of data structures offering convenient methods to access and alter data items and as well as methods for serializing and parsing to and from several output formats. Figure 2-1 shows an example of the data structure.

```
message PBHeader {
        required uint32 magic_number = 1;
        required uint32 version_major = 2;
        required uint32 version_minor = 3;
}

message PBRSSISet {

        message PBSensorData {
                optional bytes batterie_value = 1;
        }

        message PBPosition {
                optional double x_position = 1 [default = 1.79769e+308];
                optional double y_position = 2 [default = 1.79769e+308];
                optional double z_position = 3 [default = 1.79769e+308];
        }

        message PBRSSIEntry {
                required uint32 anchor_id = 1;
                required uint32 channel = 2;
                required uint32 rssi_value = 3;
```

```
            }

            required uint32 timestamp_sec = 1;
            required uint32 timestamp_usec = 2;
            required uint32 src_id = 3;
            required uint32 blink_id = 4;
            optional PBPosition position = 5;
            optional bytes payload = 6;
            optional PBSensorData sensor_data = 7;
            repeated PBRSSIEntry rssi_entry = 8;
    }
```

Figure 2-1 Format specification of the RSSI port interface

3. Getting started

The installer for Google protocol buffers, for Windows or Unix, can be found under the link https://developers.google.com/protocol-buffers/docs/downloads. The code generation version used at nanotron is version 2.4.1. Extract it to your hard disk and follow the instructions in the README file. It will instruct you on how to install it.
To start with the code generation first you need to save the protocol description file provided by Nanotron, filename.protoc, in your hard disk. Then depending on the programing language you use you will have to follow one of the three paragraphs below.

3.1. C++

Invoke the code generator by typing

```
$ protoc -I=$SRC_DIR --cpp_out=$DST_DIR $SRC_DIR/filename.proto
```

on the command line, where $SRC_DIR is the directory where your application's source code lives and $DST_DIR is the directory where you want the generated code to go (often the same as $SRC_DIR). This will create the files filename_pb.h and filename.pb.cc that contain class definitions that resemble the messages PBHeader and PBInformationSet.
The simplest way to generate the new file is to save the .proto file in the working directory, and generate the file filename_pb.h in that same directory. Then the command line should be:

```
$ protoc -I=./ --cpp_out=./ ./filename.proto
```

3.2. Python

Invoke the code generator by typing

```
$ protoc -I=$SRC_DIR --python_out=$DST_DIR $SRC_DIR/filename.proto
```

on the command line, where $SRC_DIR is the directory where your application's source code lives and $DST_DIR is the directory where you want the generated code to go (often the same as $SRC_DIR). This will create file filename_pb2.py that contains class definitions that resemble the messages PBHeader and PBInformationSet.
The simplest way to generate the new file is to save the .proto file in the working directory, and generate the file filename_pb2.py in that same directory. Then the command line should be:

```
$ protoc -I=./ --python_out=./ ./filename.proto
```

3.3. Java

Invoke the code generator by typing

```
$ protoc -I=$SRC_DIR --java_out=$DST_DIR $SRC_DIR/filename.proto
```

on the command line, where $SRC_DIR is the directory where your application's source code lives and $DST_DIR is the directory where you want the generated code to go (often the same as $SRC_DIR). This will create files filename.java that contains class definitions that resemble the messages PBHeader and PBInformationSet.
The simplest way to generate the .java file is to save the .proto file in the working directory, and generate the new file in that same directory. Then the command line should be:

```
$ protoc -I=./ --java_out=./ ./filename.proto
```

4. How to access the data

Your client application should open a TCP connection to the port of the binary interface on the machine where nanoLES is running and then start reading incoming data. (The port should be given together with the protocol description file.) The first part of the streaming is the header. After this the information is received organized in successive information messages following the information set structure. TCP does not provide message separation; to solve this problem, length-prefix framing has been applied: The message size has been added to the message itself before the actual message. This means that the payload of the received TCP packets will be: (message_size + message), where the message size is 2 bytes.
The following subsection shows some examples of client applications coded in C++ and Python.

4.1. Example using Python

This example shows the format specification of the result port (TCP 3458) of nanoLES 3:

```
message PBHeader {
        required uint32 magic_number = 1;
        required uint32 version_major = 2;
        required uint32 version_minor = 3;
}

message PBResultSet {

        message PBSensorData {
                 optional bytes battery_value = 1;
        }

        message PBPosition {
                optional double x_position = 1 [default = nan];
                optional double y_position = 2 [default = nan];
                optional double z_position = 3 [default = nan];
        }

        message PBRSSIEntry {
                 required uint32 anchor_id = 1;
                 required uint32 channel = 2;
                 required double rssi_value = 3;
        }

         message PBSectionSpecific {
                 required string section_id = 1;
                 required PBPosition position_section = 2;
                 required double ambiguity_score = 3;
                 optional double location_uncertainty = 4;
                 required bool position_valid = 5;
        }

        required uint32 timestamp_sec = 1;
        required uint32 timestamp_usec = 2;
        required uint32 src_id = 3;
        required uint32 blink_id = 4;
        required PBPosition position_estimate = 5;
        required string section_id_estimate = 6;
        optional bytes payload = 7;
        optional PBSensorData sensor_data = 8;
        repeated PBRSSIEntry rssi_entry = 9;
        repeated PBSectionSpecific section_entry = 10;
}
```

Figure 4-1 Format specification of Ressult interface

In order to access and interpret the data coming at the port we should
- open a connection at the required port
- read first 2 bytes indicating the length, L
- read next L bytes and interpret them as header
- keep on repeating the previous two steps but interpreting the information as information set.

Figure 4-1 shows and example coded in Python of how to read the data at the result port of nanoLES3.

```
import rawresultinterface10_pb2 as result
import sys
import socket
import struct

def connect (localhost,port):
    s = socket.socket()
    s.connect((localhost,port))
    return s

def read_port(s):
    data = s.recv(2)
    length, = struct.unpack('!H', data)
    data = s.recv(length)
    return data

def decode_data(first,message):
    if(first):
        header = result.PBHeader()
        header.ParseFromString(message)
        print('{0}, version
            {1}.{2}'.format(header.magic_number,header.version_major,header.version_minor))
        first = 0

    else:
        msg = result.PBResultSet()
        msg.ParseFromString(message)
        position_estimate = msg.position_estimate
        print('{0}, {1}, {2}, {3}, {4}, {5}, {6},
            {7}'.format(msg.timestamp_sec,msg.timestamp_usec,msg.src_id, msg.blink_id,
            position_estimate.x_position,position_estimate.y_position,position_estimate.z_position,
            msg.section_id_estimate)),
        for rssi_entry in msg.rssi_entry:
          print(', {0} {1} {2}'.format(rssi_entry.anchor_id,rssi_entry.channel,rssi_entry.rssi_value)),
        for section_entry in msg.section_entry:
            position_section = section_entry.position_section
            print(', {0} {1} {2} {3} {4} {5} {6}'.format(section_entry.section_id,
              position_section.x_position,position_section.y_position,position_section.z_position,
              section_entry.ambiguity_score,section_entry.location_uncertainty,section_entry.position
              _valid)),
        if msg.HasField('sensor_data'):
            print(', sensor data {0}'.format(msg.sensor_data)),
        if msg.HasField('payload'):
            print(', payload {0}'.format(msg.payload)),
        print (" ")

return first


if __name__ == '__main__':
    s = connect('127.0.0.1',3458)

    first = 1
    while(1):
        data = read_port(s)
        first = decode_data(first,data)

    s.close()
```

Figure 4-2 Python code in the application client to interpret data at the Result port

4.2. Example using C++

The following example shows how to interpret the data once it is extracted from at the RSSI port of nanoLES 2 (TCP port 3458). The structure of this interface is shown in Figure 2-1.

```
  PBRSSISet rssiSet;
  PBHeader header;
  if (first_) {
    if (!header.ParseFromString(stdString)) {
      cout << "Failed to parse header.\n";
    } else {
      cout << header.magic_number() << "; " << header.version_major()
           << "." << header.version_minor() << "\n";
    }
    first_=false;
  } else {
    if (!rssiSet.ParseFromString(stdString)) {
      cout << "Failed to parse RSSISet.\n";
    } else {
      cout << rssiSet.timestamp_sec() << ".";
      if (rssiSet.timestamp_usec() > 99)
        cout << rssiSet.timestamp_usec();
      else
        cout << "0" << rssiSet.timestamp_usec();

      cout << ";" << rssiSet.src_id() << ";" << rssiSet.blink_id();

      if(rssiSet.has_position()) {
        if(rssiSet.position().has_x_position())
          cout << ";" << rssiSet.position().x_position();
        else
          cout << ";inf";

        if(rssiSet.position().has_y_position())
          cout << ";" <<rssiSet.position().y_position();
        else
          cout << "inf;";

        if(rssiSet.position().has_z_position())
          cout << ";" <<rssiSet.position().z_position();
        else
          cout << ";inf";
      } else {
        cout << ";;;NoPosition";
      }

      if(rssiSet.has_sensor_data()) {
        if (rssiSet.sensor_data().has_batterie_value()) {
          std::string test = rssiSet.sensor_data().batterie_value();
          QByteArray arraytest(test.data(), test.size());
          QStringList list;
          list << arraytest.toHex();
          QByteArray result = list.join(",").toUtf8();
          std::string res(result.data(),result.size());
          cout << ";" << res.c_str();

        } else {
          cout << ";inf";
        }
      } else {
        cout << ";inf";
      }

      if(rssiSet.has_payload()) {
        std::string test2 = rssiSet.payload();
        QByteArray arraytest2(test2.data(), test2.size());
        QStringList list2;
        list2 << arraytest2.toHex();
        QByteArray result2 = list2.join(",").toUtf8();
        std::string res2(result2.data(),result2.size());
        cout << ";" << res2.c_str();
      } else {
        cout << ";noPayload";
      }

      for (int i = 0; i <  rssiSet.rssi_entry_size(); i++) {
        const PBRSSISet_PBRSSIEntry &rssiEntry = rssiSet.rssi_entry(i);
        cout << ";" << rssiEntry.anchor_id() << ":" << rssiEntry.channel()
             << "=" << rssiEntry.rssi_value();
      }

      cout << "\n";
    }
  }
```

Figure 4-3 C++ code in the application client to interpret data at the RSSI port

5. Available binary interfaces

RSSI interface

Available for nanoLES 2 at the TCP port 3458.
For every blink the RSSI Set shows

- time information: timestamp_sec and timestamp_usec
- blink information: ID of the tag that generated the blink (src_id) and blink sequence number (blink_id)
- optional infomation:
    - Estimated position of the tag blinking (if it was possible for nanoLES to estimate it)
    - Payload, if present in the blink
    - Sensor data: battery value (if present in the blink)
    - RSSI information: for each antenna, belonging to an anchor, detecting the blink the following information can be read:
        - ID of the anchor
        - antenna detecting the blink (channel)
        - RSSI value

Result interface

Available for nanoLES 3 at the TCP port 3458.
For every blink Result Set shows

- time information: timestamp_sec and timestamp_usec
- blink information:
    - ID of the tag that generated the blink (src_id)
    - blink sequence number (blink_id)
- position information:
    - section in which the tag is blinking (section_ID_estimate)
    - estimated position in said section (position_estimate)
- optional information:
    - Payload, if present in the blink
    - Sensor data: battery value, if present in the blink
    - RSSI information: for each antenna, belonging to an anchor, detecting the blink the following information can be read:
        - ID of the anchor
        - antenna detecting the blink (channel)
        - RSSI value
    - Sections information: for each section in which the blink is detected:
        - Section ID
        - Position inside that section (this is the position the that would have in case the tag actually was in that section)
        - Ambiguity score: score used to select what is the section in which the tag actually is.
        - Data on how good the estimation is: uncertainty of the estimation and whether the given position is actually the estimated one or had to be modified due to the presence of boundaries.

Document History

Date	Author	Version	Description
8/31/14	MLA	1.0	Guide about how to use Google Protocol Buffers
8/5/16	MLA	1.1	Minor update
13/9/16	MLA	1.2	Small clarification added

Life Support Policy

About Nanotron Technologies GmbH

Today nanotron's *embedded location platform* delivers location-awareness for safety and productivity solutions across industrial and consumer markets. The platform consists of chips, modules and software that enable precise real-time positioning and concurrent wireless communication. The ubiquitous proliferation of interoperable location platforms is creating the location-aware Internet of Things.

Further Information

For more information about products from nanotron Technologies GmbH, contact a sales representative at the following address:

nanotron Technologies GmbH
Alt-Moabit 60
10555 Berlin, Germany
Phone: +49 30 399 954 – 0
Fax: +49 30 399 954 – 188
Email: sales@nanotron.com
Internet: www.nanotron.com