Generating Track Message Data

How to use the SPx library to generate track data as UDP messages on the network.

Summary

It is sometimes necessary or desirable to generate track messages without using SPx Server or SPx Radar Simulator.

For example, to translate existing track information into a format compatible with standard SPx applications, such as SPx Radar Simulator or SPx Fusion Server.

This application note explains how such data may be generated and send onto the network, using standard SPx modules.

Overview

Generating track messages in a format that SPx applications can then receive and process or display may be useful in situations where track data exists in another format and needs to be translated into a format that SPx can accept.

Using standard modules available within the SPx library it is straightforward to generate and send valid SPx track messages onto the network. The steps involved are as follows:

1. Create an SPx packet sender object to handle packetisation and distribution of track messages.
2. Populate a suitable SPx track structure with appropriate data values, each time a new track message is created or an existing track is updated.
3. Form a track message from the SPx track structure created/updated above.
4. Pass the track message into the packet sender for dispatch onto the network.

The SPx library provides a static function, `SPxRemoteServer::TrackPackNet()`, which may be used to pack the SPx track structure into a message buffer, suitable for sending by the packet sender. This function handles byte-swapping, if appropriate, to form the data into network (big-endian) order.

The packet sender object handles all aspects of forming the message data into a packet and sending that packet onto the network. A single packet sender object should be created to provide a stream of track data and persist for the duration of the program. This object is simply created by the following line of code:

```
SPxPacketSender* TrackSender = new SPxPacketSender();
```

The address that track messages are then output to is simply set using the `SPxPacketSender::SetAddress()` function.

For example, the following line of code will configure the packet sender object to output messages to multicast address 239.192.50.79 and port number 5079:

```
TrackSender->SetAddress("239.192.50.79", 5079);
```

Filling a Structure

An SPx track structure should be filled each time a new track message is to be sent via the packet sender. The SPx track format supports varying levels of information within a track message, with options for "minimal", "normal" and "extended" messages, each containing successively more information. It is generally desirable to keep the packet as small as possible, while still conveying all of the relevant/required information. In many cases the SPx "minimal" track report will contain sufficient information, providing a unique track ID, position and velocity.

The SPx "minimal" track message format is built from an `SPxPacketTrackMinimal` C structure, defined in the "SPxPackets.h" header file as follows:

```
/* Minimal track reports. */
typedef struct SPxPacketTrackMinimal_tag {
 UINT32 id; /* Track ID (public) */
 UINT8 senderID; /* Sender identification */
 UINT8 status; /* Track status (provisional, confirmed etc.)*/
 UINT8 numCoasts; /* Number of consecutive coasts */
 UINT8 id_ttm; /* TTM id (or 0 if no ID) */
 REAL32 rangeMetres; /* Tracked Range */
 REAL32 azimuthDegrees; /* Tracked Azimuth */
 REAL32 speedMps; /* Speed */
 REAL32 courseDegrees; /* Course */
 REAL32 sizeMetres; /* Smoothed size in metres */
 REAL32 sizeDegrees; /* Smoothed size in degrees */
 UINT32 weight; /* Weight of target (number of samples) */
 UINT32 strength; /* Strength of target (sum of values) */
 UINT8 flags; /* Target flags (SPX_PACKET_TRACK_FLAGS …) */
 UINT8 validity;       /* Typically zero, or one of SPX_PACKET_TRACK_INVALID… */
 UINT8 uniTrackType; /* Typically zero, or one of SPxUniTrack::TrackType_t */
 UINT8 sourceIndex; /* Typically zero */
 UINT32 reserved3; /* Reserved, set to zero */
 UINT32 reserved4; /* Reserved, set to zero */
 UINT32 reserved5; /* Reserved, set to zero */
} SPxPacketTrackMinimal;
```

As a minimum, the ID, range, azimuth, speed and course fields should be correctly filled with appropriate values, for each update of a given track. The status field should also normally be set to a value of `SPX_PACKET_TRACK_STATUS_ESTABLISHED` to indicate that the track is in the confirmed state. For example:

```
SPxPacketTrackMinimal track = {0}; /* create a new, empty structure */
track.id = 1234; /* set appropriate unique ID */
track.rangeMetres = 5000; /* set current range, in metres */
track.azimuthDegrees = 180; /* set current azimuth, in degrees */
track.speedMps = 30; /* set current speed, in metres per second */
track.courseDegrees = 90; /* set current course, in degrees */
track.status = SPX_PACKET_TRACK_STATUS_ESTABLISHED; /* set status to confirmed */
```

The ID field uniquely determines the physical target. In order to update an existing track, successive messages should simply be generated and sent that have the same value of ID.

Forming a Message

With an appropriate SPx track data structure filled, the next step is to pack this into a message that can then be sent as the payload of a UDP packet. The `SPxRemoteServer` class provides the static `TrackPackNet()` function, which may be used for this purpose. This function takes in a pointer to a minimal (plus normal and extended) structure and packs the data into a message buffer.

The code below demonstrates the use of the `TrackPackNet()` function to pack a minimal track structure:

```
const int bufsz = sizeof(SPxPacketTrackMinimal);
unsigned char* buf = new unsigned char[bufsz];
/*
 * Pack the data into a message buffer
 * (second and third arguments are pointers to "normal" and "extended" structures,
 *  simply set to NULL if just creating a minimal track report).
 * Returns the number of bytes in the packed message.
 */
unsigned int msgBytes = SPxRemoteServer::TrackPackNet(&track,NULL,NULL,buf,bufsz);
```

Sending Messages

Now that the structure has been put into a suitable buffer, the packet sender object can be employed to dispatch the data onto the network. The SPx packet sender object provides a convenient function for this purpose, called `SendPacketB()`. This function handles the addition of an SPx packet header (which includes a timestamp) to the front of the track data payload.

The code below demonstrates how to mark the current time and dispatch a track message, where `buf` is a suitable message buffer created by the `TrackPackNet()` function.

```
SPxTime_t now;
SPxTimeGetEpoch(&now); /* Mark the current time */
sender.SendPacketB(SPX_PACKET_TYPEB_TRACK_MIN, now, buf, msgBytes);
```

Normal and Extended Track Reports

When information beyond that supported by the minimal track report needs to be supplied, a normal or extended track report may be used instead. For example, a normal track report can include track classification, and an extended track report can include many other data items including track position in latitude and longitude, secondary information from AIS, ADS-B or IFF source and so on. Track reports are nested, such that an extended report contains a normal report and a normal report in turn contains a minimal report. The above examples can be extended in this way. Note that a full definition of the contents of the normal and extended track reports is found in the file SPxPackets.h.

```
SPxPacketTrackExtended ext = {0}; /* create a new, empty structure */
SPxPacketTrackNormal* norm = &ext.norm;
SPxPacketTrackMinimal* min = &ext.norm.min;

min->id = 1234; /* set appropriate unique ID */
min->rangeMetres = 5000; /* set current range, in metres */
min->azimuthDegrees = 180; /* set current azimuth, in degrees */
min->speedMps = 30; /* set current speed, in metres per second */
min->courseDegrees = 90; /* set current course, in degrees */
min->status = SPX_PACKET_TRACK_STATUS_ESTABLISHED; /* set status to confirmed */

norm->class = 0; /* set target classification */
norm->trackClassType = SPX_PACKET_TRACK_CLASS_TYPE_CUSTOM;

ext.mask |= SPX_PACKET_TRACK_EXT_LATLONG;
ext.latDegs = 52.0; /* set target position */
ext.longDegs = 1.3;

const int bufsz = sizeof(SPxPacketTrackExtended);
unsigned char* buf = new unsigned char[bufsz];
unsigned int msgBytes = SPxRemoteServer::TrackPackNet(NULL,NULL,&ext,buf,bufsz);
SPxTime_t now;
SPxTimeGetEpoch(&now); /* Mark the current time */
sender.SendPacketB(SPX_PACKET_TYPEB_TRACK_EXT, now, buf, msgBytes);
```

Other Formats

It is straightforward to generate ASTERIX CAT-48 tracks instead of SPx format tracks. The SPx library provides an ASTERIX encoder class that may be used to convert an SPx track data structure into an ASTERIX message buffer, which may be sent onto the network via an SPx packet sender object, as before.

```
/* Make a new ASTERIX encoder object, to convert SPx track structure
 * to ASTERIX message data.
 */
SPxAsterixEncoder *AsterixEncoder = new SPxAsterixEncoder();
```

In the following code sample the `SPxPacketTrackMinimal` structure is converted into an ASTERIX message buffer, using the `BuildCat48()` function. Note that the timestamp information is included in this step, rather than at the message sending stage.

```
/* Build Asterix track message from SPx track structure */
unsigned int msgBytes = AsterixEncoder->BuildCat48(buf, bufsz,
                                    0xFFFFFFFF, &now, &track, NULL, NULL);
```

Then the `SPxPacketSender::SendRawBuffer()` function is simply used, instead of the `SPxPacketSender::SendPacketB()` function, to send the message data onto the network:

```
/* Send ASTERIX track onto network */
TrackSender->SendRawBuffer(trackBuf, msgBytes);
```

< End of document >