Navico Proprietary **Commercial in Confidence**

Technical Paper

NavRadar – BRPC SDK

PPI class description

Document Version:

D [Release], Date: 15-Aug-2016

Abstract:

This document describes how to use PPI class to interface software with Navico Radars and to generate ready to display images. Currently, this class interacts tightly with other Radar classes.

Document:	Total:
3	**4**

Revision History:

A	07-Dec-2009	Document created	Sergio Corda
B	11-Dec-2009	Updated for Library 1.0 Release	Graeme Bell
C	13-July-2016	Update for Library 3.0 Release adding Halo radar	Kevin Peng
D	15-Aug-2016	Update for Library 3.0 Release adding pulse radar	Kevin Soole

Glossary:

NRP	NavRadar Protocol. Protocol used in NAVICO BROADBAND, HALO and 12U/6X (PULSE) radar
Spoke	Data containing the sampling of a particular angle. It contains all the information for reconstructing the exact position where data were sampled.
PPI	Plan Position Indicator represents the image ready to be displayed and interpreted by radar operator.

Table of Contents

1 Overview

This document describes how to use the `tPPIController` class to generate images in connection with the `tImageClient` class which is able to control the radar and receive image data.

All data structures defined and described in this document are byte aligned and in little endian format.

2 PPI image

The task of the PPI is to create a radar image from spokes coming from radar. Other functionalities it has are: colour map selection, selection of the resolution and target trail management.

A PPI image is a conversion from the polar plane to the Cartesian plane.

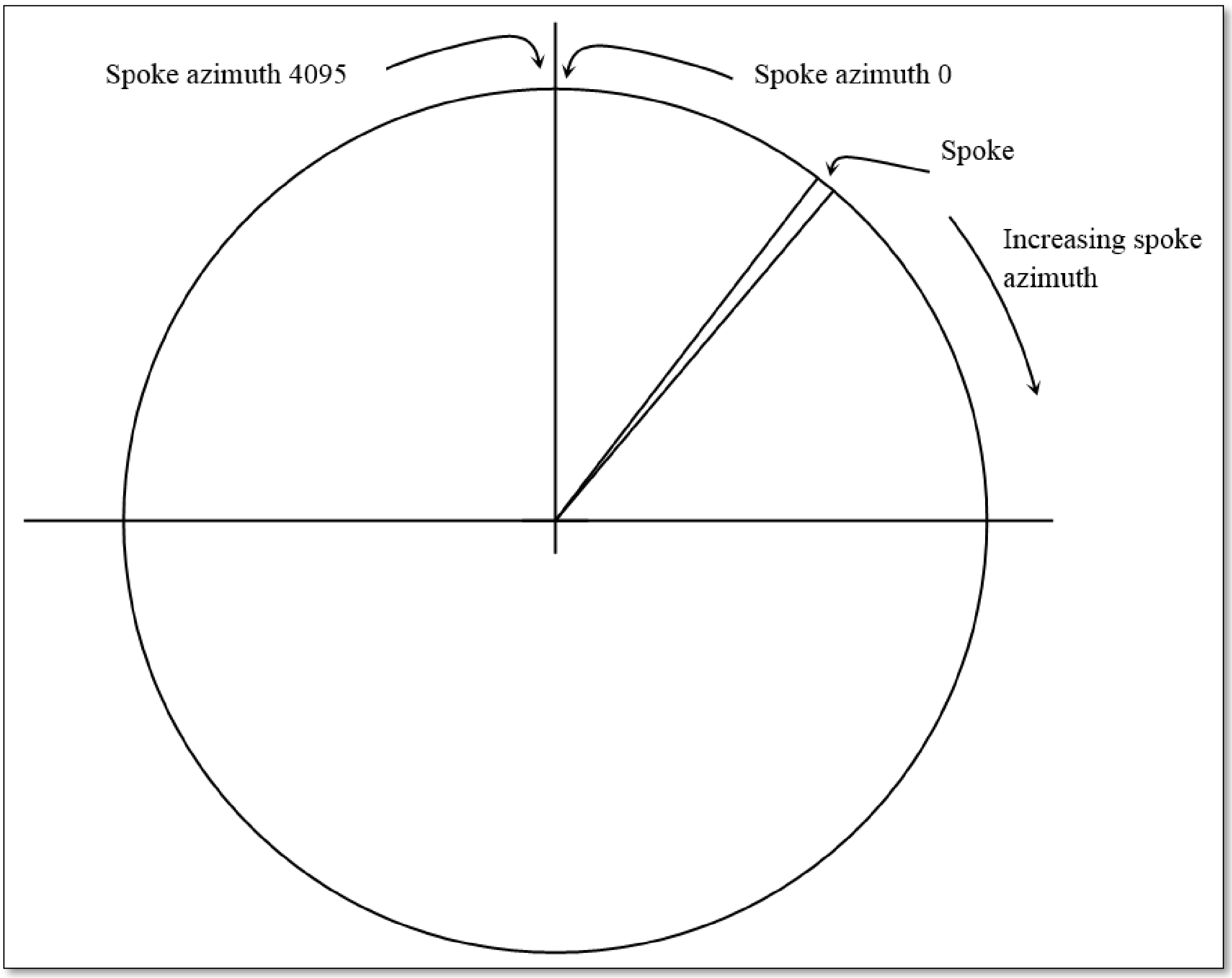

Figure 2-1: Spoke image composition

2.1 Spokes

Spokes are received by the system and processed by the PPI class to be placed in the correct position, with the correct colours and with the correct pixel trails history.

Currently, the spoke consists of a slice of cells where each cell is represented by one of 16 possible levels. The higher the level, the stronger is the target reflection for the cell. To each level a single colour can be associated.

3 Class `tPPIController`: definition and usage

The `tPPIController` class is able to draw a radar image in a frame buffer passed to this class. When the class has been set up, the drawing of each spoke will be done automatically with the simple call of a method.

example:

```
#include <PPIController.h>

#define NUM_OF_SAMPLES 1024


Navico::tRadarColourLookUpTableNavico gNavicoLUT;
...
uint8_t *m_pPPIImage = new uint8_t[ (2*NUM_OF_SAMPLES)*(2*NUM_OF_SAMPLES)*sizeof(tColor) ];
m_pPPIController = new Navico::Image::tPPIController();
...
m_pPPIController->SetFrameBuffer(
    (intptr_t)m_pPPIImage,
    (2*NUM_OF_SAMPLES), (2*NUM_OF_SAMPLES),
    &gNavicoLUT,
    (2 * NUM_OF_SAMPLES) / 2, (2 * NUM_OF_SAMPLES) / 2 );
```

After the PPI controller object has been created and set up with correct frame buffer to operate with, a spoke can be drawn easily in this way:

```
void GUIDemo::UpdateSpoke( const Spoke::t9174Spoke *pSpoke )
{
    ...
    m_pPPIController->Process( pSpoke );
    ...
}
```

In this way the whole control in the application side and correct multi threading synchronisation can be put in place, if needed.

3.1 PPI Functionality

Here the PPI classes complete functionality is described.

3.1.1 tPPIAzimuthInterpolation GetAzimuthInterpolation(void)

This command returns the kind of azimuth interpolation being used (interpolation in angle). Currently the following interpolation methods are defined:

```
enum tPPIAzimuthInterpolation
{
    eAzimuthInterpolationNone,
    eAzimuthInterpolationRepeat,
};
```

`eAzimuthInterpolationNone`: means no interpolation will be done, and the image might have some missing spokes (missing slice effect in angle) .

`eAzimuthInterpolationRepeat`: means a copy of the last spoke will be used to fill the gaps.

3.1.2 tPPIRangeInterpolation GetRangeInterpolation(void)

This command returns the kind of range interpolation being used (interpolation in range, i.e. distance from the centre). Currently the following interpolation methods are defined:

```
enum tPPIRangeInterpolation
{
    eRangeInterpolationNone,
    eRangeInterpolationPeak,
};
```

`eRangeInterpolationNone`: means no interpolation will be done, and the image might have some missing range cell data (the worst effect is to have a circle of missing data) .

`eRangeInterpolationPeak`: means it will keep the strongest target if two cells need to be drawn in the same pixel, or that the same cell will be linearly interpolated to fill possible gaps.

3.1.3 float GetRangeResolution(void)

This command will return the resolution `tPPIController` is using. This value is expressed in meters per pixel (how many meters a pixel is covering).

3.1.4 bool GetTrailsOn(void)

This command will return if trails are enabled, so that the effect of fading out of old targets is being applied.

3.1.5 int32_t GetTrailsTime(void)

This command will return the length in seconds of the trails time. The following values are defined:

0: disabled (no trails)

-1: Infinite time

>0: trails will last for this amount of seconds

3.1.6 void GetVersion(uint32_t &major, uint32_t &minor, uint32_t &build)

This command will extract the library version, providing the major, minor and build numbers.

The library version should be interpreted as: major##.minor##.build##

3.1.7 void Process(const Navico::Protocol::NRP::Spoke::t9174Spoke *pSpoke)

This command will draw a new spoke (`pSpoke` coming directly from the `tImageClient` class) in the registered frame buffer.

3.1.8 void SetAzimuthInterpolation(tPPIAzimuthInterpolation azimutInterp)

This command selects the new azimuth interpolation that will be used by the PPI controller to cover missing spokes. Currently the following interpolation methods are defined:

```
enum tPPIAzimuthInterpolation
{
    eAzimuthInterpolationNone,
    eAzimuthInterpolationRepeat,
};
```

`eAzimuthInterpolationNone`: means no interpolation will be done, and the image might have some missing spokes (missing slice effect in angle) .

`eAzimuthInterpolationRepeat`: means a copy of the last spoke will be used.

If `eAzimuthInterpolationNone` is applied, some small sectors could be missed in the final image.

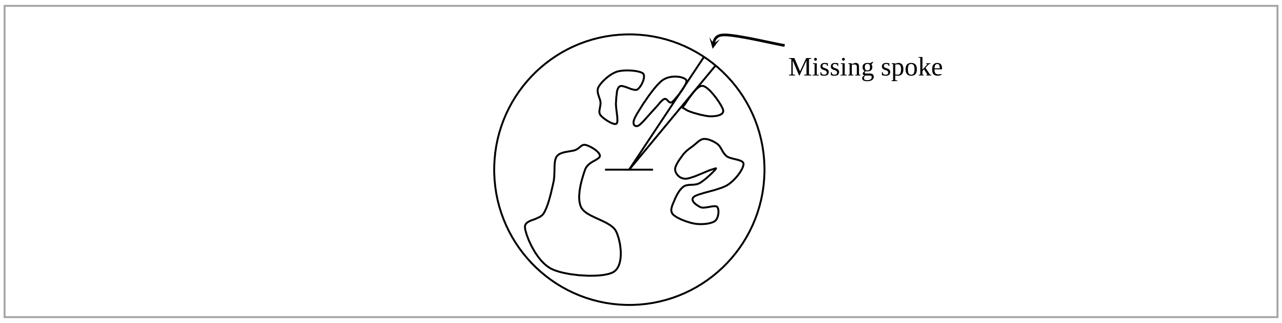

Figure 3-1: Possible effects of missing azimuth interpolation.

3.1.9 void SetColourLookUpTable(tRadarColourLookUpTable *pLookUpTable)

This command sets the colour look up table (colour map) used for drawing the radar image. It contains all the information to associate a level to a particular colour and to manage also the decaying trail effect. See the `tRadarColourLookUpTable` class definition to establish how to define a colour map.

3.1.10 void SetFramebuffer(intptr_t pBuffer, uint32_t width, uint32_t height, tRadarColourLookUpTable *pLookUpTable, uint32_t xCentre, uint32_t yCentre)

This command sets the frame buffer information to be used by the `tPPIController` class to draw the PPI.

Field	Description
`pBuffer`	is the pointer to the memory zone where the PPI will be drawn
`width`	is the width in pixels of the image
`height`	is the height in pixels of the image
`pLookUpTable`	is the pointer to the selected look up table colour map
`xCentre`	is the x centre coordinate expressed in pixels (normally it is set to `width/2`)
`yCentre`	is the y centre coordinate expressed in pixels (normally it is set to `height/2`)

A pixel width is 4 bytes wide.

3.1.11 void SetRangeInterpolation(tPPIRangeInterpolation rangeInterpolation)

This command selects the new range interpolation method to be used by the PPI controller. Currently the following interpolation methods are defined:

```
enum tPPIRangeInterpolation
{
    eRangeInterpolationNone,
    eRangeInterpolationPeak,
};
```

`eRangeInterpolationNone`: means no interpolation will be done, and the image will not be target safe (in the polar to cartesian projection some real target might be overriden by a less strong target, hiding it)

`eRangeInterpolationPeak`: means tha resulting image will be target safe, so when targets are projected from polar to cartesian coordinates, and target overlapping happens, the strongest target will be kept.

3.1.12 void SetRangeResolution(float metersPerPixel)

This command will set the resolution for the `tPPIController`. Practically sets how many meters a single pixel will represent. This parameter can be used to produce zoom effects.

3.1.13 void SetTrailsTime(int32_t time_sec)

This command sets the length in time of the trails.

The trails represent the history of targets, reporting if a target was in a particular position and how long ago the target disappeared. Practically, if a target disappears from a particular position we have a colour fading effect. The fading colours are strictly related to the colour look up table.

The following values can be passed:

0: disabled (no trails)

-1: Infinite time

>0: trails will last for this amount of seconds

3.2 Colours

Here we define how colours are managed by the PPI controller.

A target level can assume 16 different levels. To any of those levels, is associated a colour in the ARGB space (the space depends on final platforms colour management).

Each level can have 16 levels of fading colour used for the trail management.

The colour map is defined as follows:

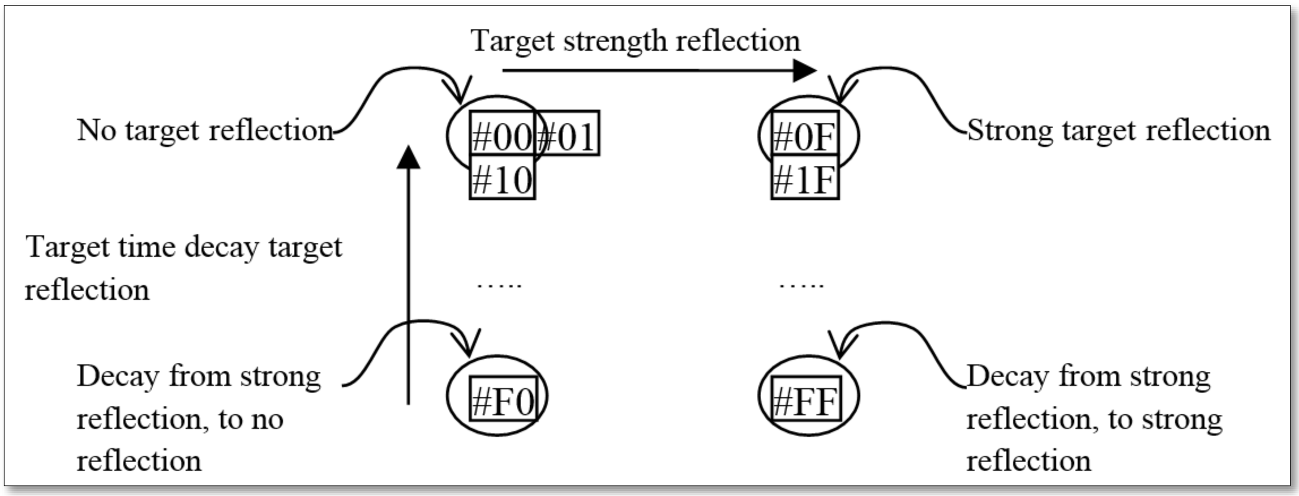
Target strength reflection
No target reflection
#00
#01
#10
#0F
#1F
Strong target reflection
Target time decay target reflection
.....
.....
Decay from strong reflection, to no reflection
#F0
#FF
Decay from strong reflection, to strong reflection

Figure 3-2: Colour map representation.

For example, if we consider a 2 dimensional matrix where the first index is the `targetStrength` and the second is the `targetDecay` and we consider that a particular target in a particular location during consecutive scans moves between the levels

`0xf`, `0x0`, `0x0`, `0x0`, the colour element used will be: `0xff`, `0xe0`, `0xd0`, `0xc0`

For different scans we could have

`0xf`, `0x9`, `0x0`, `0x9`, the colour element used will be: `0xff`, `0xe9`, `0xd0`, `0xc9`

3.2.1 Colour map definition

A new colour map shall be defined with a matrix of colours and with a class derived by `tRadarColourLookUpTable`.

example:

```
#include "RadarColourLookUpTable.h"

using namespace Navico;
...

class tRadarColourLookUpTableUser: public tRadarColourLookUpTable
{
public:
    tRadarColourLookUpTableUser();
};

...
tColor LUTUser[256] =
{
    FromARGB( ALPHA_VALUE, 0x00, 0x00, 0x00 ), FromARGB( ALPHA_VALUE, 0x00, 0x10, 0x00 ),
FromARGB( ALPHA_VALUE, 0x00, 0x20, 0x00 ), FromARGB( ALPHA_VALUE, 0x00, 0x30, 0x00 ),
FromARGB( ALPHA_VALUE, 0x00, 0x40, 0x00 ), FromARGB( ALPHA_VALUE, 0x00, 0x50, 0x00 ),
FromARGB( ALPHA_VALUE, 0x00, 0x60, 0x00 ), FromARGB( ALPHA_VALUE, 0x00, 0x70, 0x00 ),
FromARGB( ALPHA_VALUE, 0x00, 0x80, 0x00 ), FromARGB( ALPHA_VALUE, 0x00, 0x90, 0x00 ),
FromARGB( ALPHA_VALUE, 0x00, 0xa0, 0x00 ), FromARGB( ALPHA_VALUE, 0x00, 0xb0, 0x00 ),
FromARGB( ALPHA_VALUE, 0x00, 0xc0, 0x00 ), FromARGB( ALPHA_VALUE, 0x00, 0xd0, 0x00 ),
FromARGB( ALPHA_VALUE, 0x00, 0xe0, 0x00 ), FromARGB( ALPHA_VALUE, 0x00, 0xf0, 0x00 ),
    ...
    FromARGB( ALPHA_VALUE, 0xff, 0xff, 0xff ), FromARGB( ALPHA_VALUE, 0xff, 0x00, 0xff ),
FromARGB( ALPHA_VALUE, 0xff, 0x00, 0xff ), FromARGB( ALPHA_VALUE, 0xff, 0x00, 0xff ),
FromARGB( ALPHA_VALUE, 0xff, 0x00, 0xff ), FromARGB( ALPHA_VALUE, 0xff, 0x00, 0xff ),
FromARGB( ALPHA_VALUE, 0xff, 0x00, 0xff ), FromARGB( ALPHA_VALUE, 0xff, 0x00, 0xff ),
FromARGB( ALPHA_VALUE, 0x00, 0x80, 0x00 ), FromARGB( ALPHA_VALUE, 0x00, 0x90, 0x00 ),
FromARGB( ALPHA_VALUE, 0x00, 0xa0, 0x00 ), FromARGB( ALPHA_VALUE, 0x00, 0xb0, 0x00 ),
FromARGB( ALPHA_VALUE, 0x00, 0xc0, 0x00 ), FromARGB( ALPHA_VALUE, 0x00, 0xd0, 0x00 ),
FromARGB( ALPHA_VALUE, 0x00, 0xe0, 0x00 ), FromARGB( ALPHA_VALUE, 0x00, 0xf0, 0x00 ),
};

...
tRadarColourLookUpTableUser::tRadarColourLookUpTableUser()
{
    SetLookUpTablePointer( LUTUser );
};
```

Now the `tRadarColourLookUpTableUser` colour map has been defined and can be used by the `tPPIController`.