Technical Paper

NavRadar – BRPC SDK

NRP class description

Document Version:

E [Release], Date: 05-Sept-2016

Abstract:

This document describes how to use NRP Image-Client Class to interface software with Navico Radars.

Document:	Total:
1	4

Revision History:

A	02-Jun-2009	Document created	Sergio Corda
B	11-Dec-2009	Updated for Library 1.0 Release	Graeme Bell
C	21-Sept-2011	Updated for Library 2.0 Release	Graeme Bell
D	02-July-2016	Update for Library 3.0 Release adding Halo radar	Kevin Peng
E	05-Sep-2016	Update for Library 3.0 Release adding Pulse radar	Kevin Soole

Glossary:

NRP	NavRadar Protocol. Protocol used in NAVICO BROADBAND, PULSE COMRESSION and PULSE radars
Spoke	Data containing the sampling of a particular angle. It contains all the information for reconstructing the exact position where data were sampled
CCRP	Common Consistent Reference Point
Lock-ID	Unique radar capability identifier required to acquire an Unlock-Key
Unlock-Key	A key that can be used to unlock a radar
FMCW	Frequency Modulated Continuous Wave This is the technology used in Navico BROADBAND radars (BR24, 3G, 4G)
Pulse Compression	This is the technology used in Navico PULSE COMRESSION radars (HALO)
Pulse	This is the technology used in Navico PULSE radars (12U/6X)

Table of Contents

1. Overview

This document describes how to use the `tImageClient` class to connect to a radar, and perform simple operations such as changing range, changing gain, and getting data from the radar to display an image.

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

1.1. Lock Levels

The BRPC SDK contains software libraries that allows partner companies to develop software to integrate Navico Radars into their own systems

An Unlock Key is required to enable a radar to communicate with the SDK. Unlock Keys are personalized to each individual Radar and must be purchased from Navico

A separate Unlock Key is required for each and every radar, therefore, two Radar's requires the purchase of two Unlock Keys

Unlock Keys are available based on varying functionality levels. See the separate **BRPC SDK Controls** document for information on Lock Level functionality (LL1, LL2 etc).

Contact your Navico BRPC Representative for pricing and to purchase Unlock Keys

1.2. Radar Functionality

Navico provide Marine Radars with a range of technologies to suit many different applications. However, not all controls provided in this SDK maybe supported by each radar. See the separate **BRPC SDK Controls** document for information on controls supported by radar.

2. Radar States

The radar has six different states it can be in. The following diagram lists the possible states it can assume and how it transitions between them. Some radars require minimum or no Detect Scanner and / or Warming Up before reaching the Stand By state.

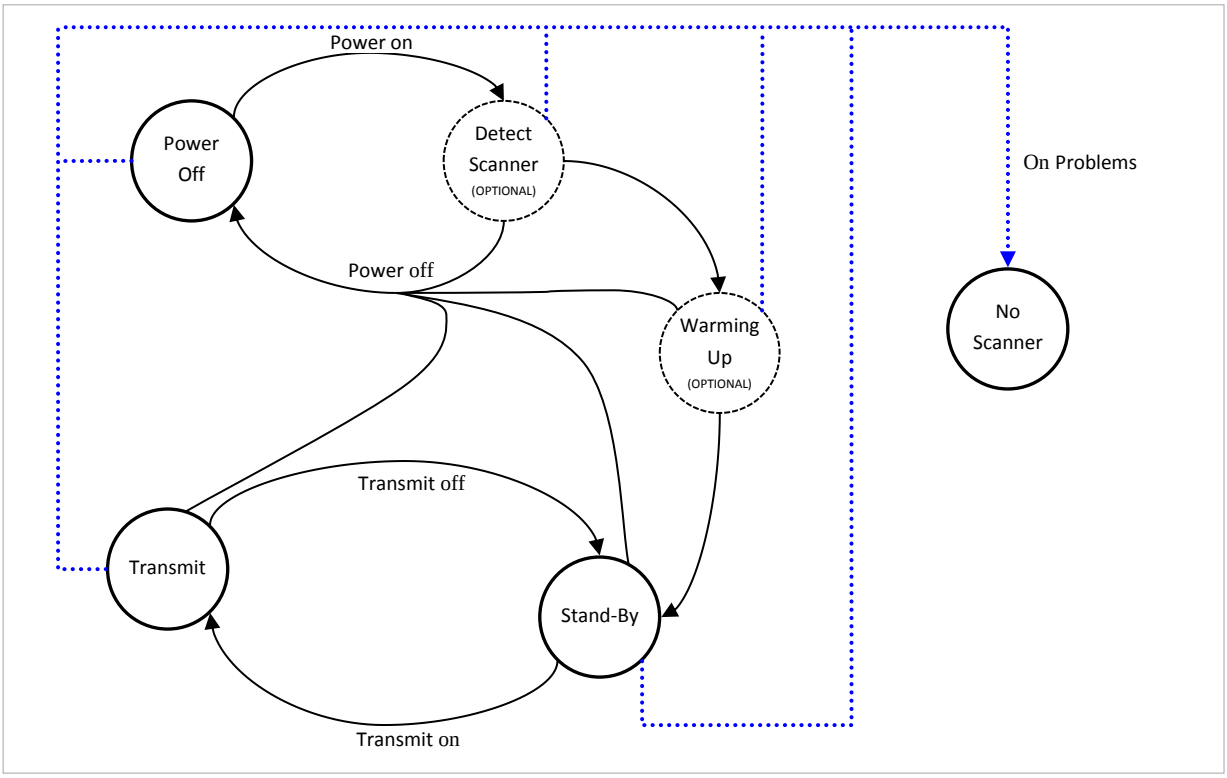

Figure 2-1: Radar state machine

Note that for radars supporting multiple image-streams/ranges any stream transitioning to the *power-off* state will cause all streams to transition to the *power-off* state.

2.1. RPM Limits

Although you may select above normal rotation rates for the antenna various factors can stop the radar using the requested rate. One of the main ones is range; faster rotation rates are only supported at closer ranges, often less than 2 nautical miles.

Transmitting on multiple image-streams of the same radar at the same time will also restrict the RPM to normal, as can some signal-processing controls such as noise-reject.

2.2. STC Curves

The radar uses a combination of three separate STC (Sensitivity-Time-Constant) curves to compensate for the various types of attenuation that can occur to the signal returns from targets.

Field	Curve	Description
Range STC	R^4	Attenuation due to the range of the target
Sea STC	~R^3	Attenuation due to sea clutter – reflections from the sea (actually a modified R^3)
Rain STC	R^2	Attenuation due to rain

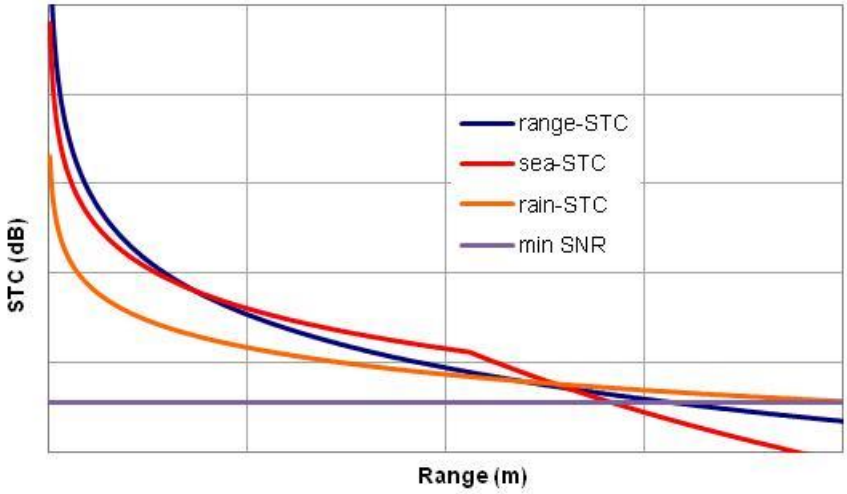

Figure 2-2: Individual STC Curves

These three curves, plus the minimum SNR value, are actually combined into a single STC curve, by taking their maximum, and this combined curve is then subtracted from the return signal.

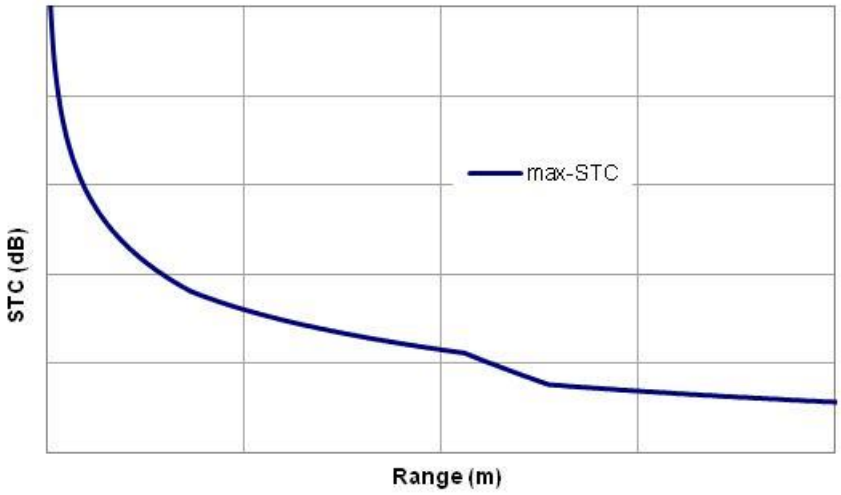

Figure 2-3: Combined STC Curve

2.3. Minimising Spoke Processing

It is not possible to get completely raw/unprocessed spokes from the Radar, but this section describes how to minimise the processing performed by the radar.

This code adjusts the radars controls to minimise its processing.

```
SetGain( eUserGainManual, 128 );      // 50% gain
SetSeaClutter( UserGainManual, 0 );   // 0% sea-clutter
SetRain( 0 );                         // 0% rain gain
SetSideLobe( eUserGainManual, 0 );    // 0% sidelobe gain
SetSTCCurveType( 1 );                 // moderate sea stc-curve
SetInterferenceReject( 0 );           // no interference-reject
SetTargetStretch( false );            // no target-stretch
SetTargetBoost( 0 );                  // no target-boost/emphasise
```

```
SetFastScanMode( 0 );                // normal speed
```

The following code uses advanced features (only available when the radar and library are unlocked for level 2) to further reduce the processing performed by the radar.

```
SetLocalIR( 0 );                     // no local interference-reject
SetNoiseReject( 0 );                 // no noise reject
SetBeamSharpening( 0 );              // no beam sharpening
SetScannerRPM( 24 );                 // 24rpm ensure normal speed
SetUserMinSNR( 0 );                  // 0dB signal to noise ratio
SetVideoAperture( 4 );               // 4dB video-aperture
```

Lastly the advanced trim and rate controls can be used to adjust some of the remaining processing to better suit your application.

3. NRP protocol

Currently the communications protocol is based on multicast UDP/IP, with each radar on the network using a separate set of multicast addresses for receiving commands and the transmission of state as well as radar image information.

3.1. Spoke distribution

Spokes (radar image data) are distributed in packets with each single packet able to contain multiple spokes. Each spoke has the following format:

```
struct t9174Spoke
{
    t9174SpokeHeader header;
    uint8_t  data[ SAMPLES_PER_SPOKE/2 ];
};
```

Normally the number of samples per spoke is set to 1024, meaning a `data` size of 512 bytes for 4-bit samples, but the actual number of samples (and their size) is contained in fields of the spoke header structure `t9174SpokeHeader`.

The spoke header is defined as follows:

```
struct t9174SpokeHeader
{
    uint32_t spokeLength_bytes : 12;          // Length of the whole spoke in bytes
    uint32_t : 4;                             // reserved
    uint32_t sequenceNumber : 12;             // Spoke sequence number
    uint32_t : 4;                             // reserved
    uint32_t nOfSamples: 12;                  // Number of samples in the spoke
    uint32_t bitsPerSize: 4;                  // Number of bits used to store each sample,
                                              // normally 4
    uint32_t rangeCellSize_mm : 16;           // Distance represented by each range-cell

    uint32_t spokeAzimuth: 13;                // Azimuth of the spoke in the range 0-4095
    uint32_t : 1;                             // reserved
    uint32_t bearingZeroError: 1;             // Set if there is a faulty bearing zero
    uint32_t : 1;                             // reserved
    uint32_t spokeCompass: 14;                // Heading of the boat when this spoke was
                                              // sampled. Its represented in the 0-4095 range
                                              // for 0-360 degrees.
    uint32_t trueNorth : 1;                   // Connected heading sensor is reporting true
                                              // north (1) or magnetic north (0).
    uint32_t compassInvalid : 1;              // If 1, the compass information is invalid
    uint32_t rangeCellsDiv2 : 16;             // Number of range-cells/2 represented by this
                                              // spoke
    uint32_t : 16;                            // reserved
    uint32_t : 16;                            // reserved
    uint32_t : 16;                            // reserved
    uint32_t : 16;                            // reserved
    uint32_t : 16;                            // reserved
};
```

These fields are described in the following table.

Field	Description
spokeLength_bytes	Length of the spoke expressed in bytes, including header and data. Normally set to 536 (size of header plus 1024 x 4-bit samples)
sequenceNumber	Sequence number of the spoke. It is a value increasing monotonically in the range 0-4095
nOfSamples	Number of samples in the spoke. Normally this is 1024
bitsPerSample	Number of bits used to store each sample. Normally set to 4 (16 colours)

rangeCellSize_mm	The distance represented by each range-cell, in millimetres. Normally it is a value between 0 and 65535. The actual distance represented by each pixel/sample can be gotten by using the `GetSampleRange_mm` function and is computed as `(rangeCellSize_mm` * 2 * `rangeCellsDiv2)` / `nOfSamples`
spokeAzimuth	The angular position of the spoke relative to the head of the boat. This is a value that normally varies in the range 0-4095. If the value is greater than 4095, the spoke should be mapped to the 4095 value. This range represents a full 0-360 degrees (ie. degrees = 360 * `spokeAzimuth` / 4096)
bearingZeroError	If this field is set to 1 it means there are problems recognising the 0 azimuth position of the radar antenna
spokeCompass	This field contains the value of the heading sensor when the spoke was sampled. If no heading sensor is connected to the radar this field meaningless. The value of this field can vary between 0-4095 representing 0-360 degrees
trueNorth	If this field is set it means that the `spokeCompass` has to be interpreted as relative to the true north, otherwise to the magnetic north
compassInvalid	If set to 1, it means that no heading sensor is connected to the radar and the field's `spokeCompass` and `trueNorth` are meaningless
rangeCellsDiv2	The total number of range-cells calculated for this spoke, divided by 2. The actual range covered by the spoke can be gotten using the `GetSpokeRange_mm` function and is computed as `(rangeCellSize_mm` * 2 * `rangeCellsDiv2)`

Table 3-1: Spoke packet header description.

Data samples, in the 4-bit format, are byte packed with the least significant bits stored the low-index samples, and in the most significant bits the high-index samples. A sample value of 0 corresponds to no signal, and a sample value of 15 to the strongest target signal captured by the radar.

Note: For Ethernet clients the over-scan from the selected instrumental range is set to 1.8 (the actual value may be affected by the current target-boost setting – see `bool` SetTargetBoost(uint8_t level)). Practically this means that samples from 0 to 569 are inside the instrumental range (see the protocol for setting the instrumental range), all the remaining pixels are the over scan data (normally used for the look-ahead mode).

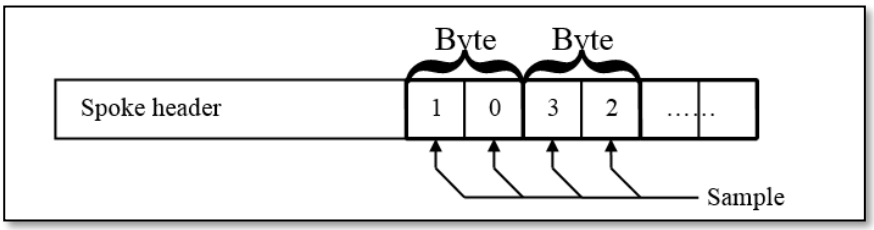

Figure 3-1: Spoke packetisation for 4-bit data samples

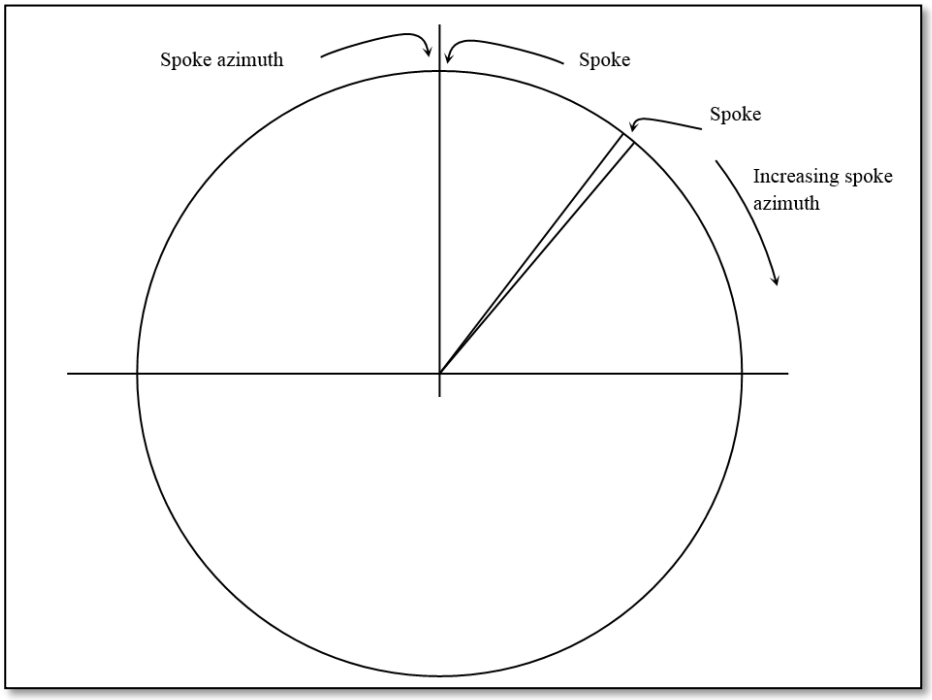

Figure 3-2: Spoke image composition

A radar image is composed of spokes that are ordered by increasing azimuth in a clockwise direction. The spoke with an azimuth of zero is oriented in the same direction as the head of the boat and indicates the beginning of a new image (scan). Normally the 4095th spoke is the last one of the scan, but in some situations a spoke can have an azimuth greater than 4095. These need to be mapped down to azimuth 4095.

3.2. Control

This channel is used to send commands to the radar for changing its setup or mode of operation. Typical operations that can be requested are: powering of the system, setting the transmission mode, changing instrumental range, tuning the gains, etc. Most changes will cause the radar to resend its current state.

3.3. State

The radar communicates range changes, operating mode changes, gain changes etc using this channel. Most changes will automatically cause the radar to resend its current state, but the information can also be directly requested using one of several query commands.

4. Class `tImageClient`: definition and usage

The `tImageClient` class is able to interface a client with a radar without the client needing to worry about the network protocol. This isolates the clients from future changes to the radar protocol.

This class works with the observer paradigm which means an object interested in receiving radar images and status/state information needs to derive from the appropriate observer classes (`iImageStateObserver`, `iImageSpokeObserver` and `iFeatureObserver`) and register itself with the `tImageClient` class.

example:

```
#include <ImageClientObserver.h>
#include <ImageClient.h>
#include <FeatureManager.h>

using Navico::Protocol;

class RadarClient
    : public NRP::iImageClientStateObserver
    , public NRP::iImageClientSpokeObserver
    , public NRP::iFeatureObserver
{
...

public:

// iImageClientSpokeObserver callbacks
        void UpdateSpoke( const NRP::Spoke::t9174Spoke* pSpoke );

// iImageClientStateObserver callbacks
        void UpdateMode( const NRP::tMode* pMode );
        void UpdateSetup( const NRP::tSetup* pSetup );
        void UpdateSetupExtended( const NRP::tSetupExtended* pSetupExtended );
        void UpdateProperties( const NRP::tProperties* pProperties );
        void UpdateConfiguration( const NRP::tConfiguration* pConfiguration );
        void UpdateGuardZoneAlarm( const NRP::tGuardZoneAlarm* pAlarm );
        void UpdateRadarError( const NRP::tRadarError* pError );
    void UpdateAdvancedState( const NRP::tAdvancedSTCState* pState );

// iFeatureObserver callbacks
        void UpdateFeature ( const NRP::tFeatureEnum* pFeature );
    ...

    private:
        NRP::tImageClient* m_pImageClient;
    };
```

Registration of observers with the `tImageClient` class can be done using these methods

```
bool tImageClient::AddSpokeObserver( iImageClientSpokeObserver *pObserver )
bool tImageClient::AddStateObserver( iImageClientStateObserver *pObserver )
bool tImageClient::AddFeatureObserver( iFeatureObserver *pObserver )
```

example:

// RadarClient

```
m_pImageClient->AddSpokeObserver( this )
m_pImageClient->AddStateObserver( this )
m_pImageClient->AddFeatureObserver( this )
```

After registration you need to connect to the radar using the following method

```
int tImageClient::Connect( const char * pSerialNumber, unsigned imageStream = 0 )
```

Providing the serial-number of the radar you wish to connect too, and which instance of its Image services. The serial-number of radars available on the network can be obtained from the `tMultiRadarClient` class, as can the number of image-streams it provides (see `tMultiRadarClient::GetImageStreamCount`).

Once you have successfully connected to the radar the `tImageClient` will be able to transmit and receive commands to and from the radar. The reception of state information will start immediately so it is possible for an update call-back to be invoked even before the `Connect` method has returned.

4.1. Commands to the radar

Below are details of the methods provided by the `tImageClient` class for interacting with a radar.

Most commands return "true" if the operation was successfully initiated. This only means that the command was successfully queued for transmission over the Ethernet, not that the command has been successfully received and processed by the radar.

In most cases the radar will automatically send a status or state update message containing the changed setup or configuration data when it receives a command. For commands that don't initiate an automatic response an appropriate Query command can be sent. This response can be used to confirm the operation has been completed successfully by the radar.

4.1.1. bool SetPower(bool state)

This command will power the radar on or off; it is not a physical power off but is related to the state of the scanner. If `state` is set to "true" the scanner power state will be set to on and the radar will move from the *power-off* to the *standby* state, passing via the *warming-up* state. A "false" value will cause the radar to transition to the *power-off* state.

This is one of the few commands that effect all image-streams; if two separate `tImageClient` instances have been connected to different image-streams on the same radar then this command will force both of them to transition to the *power-off* state.

It will return "true" if the operation was successfully initiated, "false" otherwise.

4.1.2. bool SetTransmit(bool state)

This command will start or stop the radar from transmitting spokes (radar image data). If `state` is set to "true" the scanner will move from the *stand-by* or *warming-up* state, to the *transmit* state; "false" will send the radar back to the *stand-by* state. Enabling the *transmit* state will normally start the scanner antenna rotating.

Note that this command can cause the maximum RPM rate to be restricted if multiple image-streams on the same radar are set to transmit at the same time.

It will return "true" if the operation was successfully initiated, "false" otherwise.

4.1.3. bool SetTimedTransmit(bool state)

This command puts the radar into a mode where it will sit in the *stand-by* state for a configured period of time, then transition to the *transmit* state and transmit for a second period of time before dropping back into *stand-by* state to repeat the sequence. If `state` is set to "true" timed-transmit mode will be enabled otherwise it will be disabled.

It will return "true" if the operation was successfully initiated, "false" otherwise.

4.1.4. bool SetTimedTransmitSetup(uint32_t transmitPeriod_sec, uint32_t standbyPeriod_sec)

This command configures the periods to be used when the radar is in timed-transmit mode (see the `SetTimedTransmit` method immediately above). `transmitPeriod_sec` is the number of seconds the radar should transmit for before returning to *stand-by* state, and `standbyPeriod_sec` the number of seconds it should stay in *stand-by* state before transmitting again.

It will return "true" if the operation was successfully initiated, "false" otherwise.

4.1.5. bool SetRange(uint32_t range_m)

This command will change the instrumental range of the radar. The `range_m` represents the instrumental range desired expressed in metres and must be at least 15 metres. Note that in the image data there will always be an over-scan factor of between 1.5 and 1.8, depending on the type of radar.

It will return "true" if the operation was successfully initiated, "false" otherwise.

4.1.6. bool SendClientWatchdog()

This command has to be sent every ~8 seconds. When this command has not been sent for ~30 seconds the radar will automatically move from the *transmit* state to the *stand-by* state, and after ~1 minute the radar will move to the *power-off* state. This is a safety feature for all radars. The `SetAutoSendClientWatchdog` method below may be used to enable the library to automatically send client watchdog commands.

It will return "true" if the operation was successfully initiated, "false" otherwise.

4.1.7. bool SetAutoSendClientWatchdog(bool state)

This command can be used to get the library to automatically perform a `SendClientWatchdog` every ~8 seconds. If `state` is "true" (and `Connect` has been successfully called) the library will call `SendClientWatchdog`, otherwise it must be externally invoked to ensure the radar keeps transmitting spokes.

4.1.8. bool SetUseMode(eUseMode mode)

This command sets the use mode. The mode can be any of the supported use modes reported by the radar.

It will return "true" if the operation was successfully initiated, "false" otherwise.

4.1.9. bool SetGain(eUserGainManualAuto type, uint8_t level)

This command sets the radar gain. The gain can be set as manual (`eUserGainManual`) or automatic (`eUserGainAuto`) in the `type` parameter. If it is set to manual, then the gain will be set to the `level` value. The `level` parameter can assume a value in the range 0-255., where 0 is the minimum gain and 255 the maximum.

It will return "true" if the operation was successfully initiated, "false" otherwise.

4.1.10. bool SetSeaClutter(eUserGainManualAuto type, uint8_t level)

This legacy command sets the sea clutter level in the radar. The sea clutter can be set as manual (`eUserGainManual`) or automatic (`eUserGainAutoHarbour`, or `eUserGainAutoOffshore`) in the `type` parameter. If it is set to manual, then the sea clutter will be set to the `level` value. The `level` parameter can assume a value in the range 0-255, where 0 is the minimum sea clutter rejection and 255 is the maximum.

It will return "true" if the operation was successfully initiated, "false" otherwise.

4.1.11. bool SetSeaClutter(eUserGainManualAuto type, uint8_t level, int8_t autoOffset, eUserGainValid valid)

This command sets the sea clutter level in the radar. The sea clutter can be set as manual (`eUserGainManual`) or automatic (`eUserGainAuto`) in the `type` parameter. If it is set to manual, then the sea clutter will be set to the `level` value. If it is set to automatic, the `level` value will be used as a start point and then automatically tuned based on the `autoOffset` value. The `level` parameter can assume a value in the range 0-255, where 0 is the minimum sea clutter rejection and 255 is the maximum. The `autoOffset` parameter can assume a value in the range -128-127, where -128 is the minimum offset and 127 is the maximum. The `valid` parameter is the bit mask for validity of supplied parameters, where `UserGainValid_Mode` indicates the auto/manual type shall be

updated, `UserGainValid_Level` indicates the manual level shall be updated, `UserGainValid_Offset` indicates the auto offset shall be updated, and `UserGainValid_ALL` indicates that all three parameters shall be updated.

It will return "true" if the operation was successfully initiated, "false" otherwise.

4.1.12. bool SetSideLobe(eUserGainManualAuto type, uint8_t level)

This command sets the radar side-lobe mode and level. The mode can be set as manual (`eUserGainManual`) or automatic (`eUserGainAuto`) in the `type` parameter. If it is set to manual, then the side-lobe level will be set to the `level` value. The `level` parameter can assume a value in the range 0-255., where 0 means no side-lobe processing and 255 the maximum.

It will return "true" if the operation was successfully initiated, "false" otherwise.

4.1.13. bool SetSTCCurveType(eStcCurveType type)

This command sets the STC curve type that will be applied to the automatic sea clutter rejection (modes `eUserGainAutoHarbour` and `eUserGainAutoOffshore`). The `type` parameter reflects the sea-states: `eCalm` (0), `eModerate` (1), and `eRough` (2).

It will return "true" if the operation was successfully initiated, "false" otherwise.

4.1.14. bool SetFastScanMode(uint8_t level)

This command sets the radars fast scanner rotation mode. If the `level` parameter is 0 the scanner will be set into fast scan/rotation mode, otherwise its rotation speed will be restored normal. Note that fast scan mode only operates at shorter ranges, currently this is only for ranges of 2 nautical-miles and below (see section 2.1 "RPM Limits" for other restrictions).

It will return "true" if the operation was successfully initiated, "false" otherwise.

4.1.15. bool SetRain(uint8_t level)

This command sets the rain clutter level in the radar. Rain clutter rejection can only be manual and will be set to the `level` value. The `level` parameter can assume a value in the range 0-255, where 0 is the minimum rain clutter rejection and 255 maximum.

It will return "true" if the operation was successfully initiated, "false" otherwise.

4.1.16. bool SetFTC(uint8_t level)

This command sets the FTC (Fast-Time-Constant) level in the radar. The FTC can only be manual and will be set to the `level` value. The `level` parameter can assume a value in the range 0-255. This control is also affected when the rain clutter level is adjusted.

It will return "true" if the operation was successfully initiated, "false" otherwise.

4.1.17. bool SetInterferenceReject(uint8_t level)

This command sets the interference reject level in the radar. The interference reject `level` can be a value in the range 0-3, where 0 means interference reject is disabled and 3 provides the highest level of interference rejection.

It will return "true" if the operation was successfully initiated, "false" otherwise.

4.1.18. bool SetLocalIR(uint8_t level)

This command sets the level of local interference-reject which can be used to help reduce interference caused by other on-board devices. The `level` parameter can assume values in the range 0: disabled, to 2: high.

It will return “true” if the operation was successfully initiated, “false” otherwise.

4.1.19. bool SetNoiseReject(uint8_t level)

This command sets the level of noise-reject which can be used to increase the sensitivity of the radar improving target detection but possibly reducing the sharpness of the image. It only operates at longer ranges and can restrict maximum RPM (see section 2.1 “RPM Limits”). The `level` parameter can assume values in the range 0: disabled, to 2: high.

It will return “true” if the operation was successfully initiated, “false” otherwise.

4.1.20. bool SetBeamSharpening(uint8_t level)

This command sets the level of beam-sharpening which improves the resolution of the radar image in azimuth. The `level` parameter can assume values in the range 0: disabled, to 2: high.

It will return “true” if the operation was successfully initiated, “false” otherwise.

4.1.21. bool SetLEDsLevel(uint8_t level)

This command sets the lighting level of LEDs.

It will return “true” if the operation was successfully initiated, “false” otherwise.

4.1.22. bool SetTargetStretch(bool state)

This command enables artificial (using interpolation methods) target expansion in range. It can be enabled or disabled. Note that this command was originally called `SetTargetExpansion`, it has been renamed for clarity.

It will return “true” if the operation was successfully initiated, “false” otherwise.

4.1.23. bool SetTargetStretch(uint8_t level)

This command sets the level of target stretch to apply. The level can be 0: none, to 3: high, or the maximum level reported by the radar.

It will return “true” if the operation was successfully initiated, “false” otherwise.

4.1.24. bool SetTargetBoost(uint8_t level)

This command provides another way of expanding targets, which is based on a different principle from the `SetTargetExpansion` command above. It can assume a level between 0 and 2, where 0 means target boost is disabled (better target resolution) and 2 means maximum target boost (better target emphasis).

Note that level 0 may affect the amount of over-scan on image data.

It will return “true” if the operation was successfully initiated, “false” otherwise.

4.1.25. bool QueryAll()

This command asks the radar to send back the most important “Mode” and “Setup” information. It is equivalent to performing all 5of the following query commands.

It will return “true” if the operation was successfully initiated, “false” otherwise.

4.1.26. bool QueryMode()

This command asks the radar to send all the radar "Mode" information, which should result in the `iImageClientObserver::UpdateMode` call-back being invoked. This information is also sent regularly by the radar without the need to send a specific query.

It will return "true" if the operation was successfully initiated, "false" otherwise.

4.1.27. bool QuerySetup()

This command asks the radar to send all radar "Setup" information, which should result in the `iImageClientObserver::UpdateSetup` and `iImageClientObserver::UpdateSetupExtended` call-backs being invoked.

It will return "true" if the operation was successfully initiated, "false" otherwise.

4.1.28. bool QueryAdvancedSetup()

This command asks the radar to send all advanced radar setup information, which should result in the `iImageClientObserver::UpdateAdvancedState` call-back being invoked.

It will return "true" if the operation was successfully initiated, "false" otherwise.

4.1.29. bool QueryProperties()

This command asks the radar to send all information about the radars "Properties", which should result in the `iImageClientObserver::UpdateProperties` call-back being invoked.

It will return "true" if the operation was successfully initiated, "false" otherwise.

4.1.30. bool QueryConfiguration()

This command asks the radar to send all the information about "Configuration" data, which should result in the `iImageClientObserver::UpdateConfiguration` call-back being invoked.

It will return "true" if the operation was successfully initiated, "false" otherwise.

4.1.31. bool QueryFeatures()

This command asks the radar to send all the information about "Feature" data, which should result in the `iFeatureObserver::UpdateFeature` call-back being invoked repeatedly for each feature supported by the radar.

It will return "true" if the operation was successfully initiated, "false" otherwise.

4.1.32. bool SetParkPosition(uint32_t angle_deg)

This command sets the parking angle of the radar. This is only used for open array scanners and it is ignored by some radars. `angle_deg` is the desired parking angle relative to the radars zero-bearing position, in degrees. This value will be stored internally in the radar, so it is not necessary to resend this value at each start up.

It will return "true" if the operation was successfully initiated, "false" otherwise.

4.1.33. bool SetZeroRangeOffset(uint16_t offset)

This command sets the range offset of the radar. This is used only for PULSE radars, and it is completely ignored by the BROADBAND and PULSE COMPRESSION radars. `offset` can assume a value between 0 and 1023. If supported, this value will be saved by the radar and so does not need to be resent at each start up.

It will return "true" if the operation was successfully initiated, "false" otherwise.

4.1.34. bool SetZeroBearingOffset(uint16_t bearing_ddeg)

This command sets the zero bearing offset of the radar. This can be used to correct errors due to the misalignment of the radar from the head of the boat. `bearing_ddeg` is the offset value that will be applied for correcting the zero position and it can assume a value between 0 and 3599 (10^{ths} of a degree AKA deci-degrees). This value will be saved by the radar and so does not need to be resent at each start-up.

It will return "true" if the operation was successfully initiated, "false" otherwise.

4.1.35. bool SetAntennaHeight(uint32_t antennaHeight_mm)

This command sets the antenna height. This parameter should be set for having better automatic radar image tuning. This value will be saved by the radar and so does not need to be resent at each start-up.

It will return "true" if the operation was successfully initiated, "false" otherwise.

4.1.36. bool SetAntennaOffsets (int32_t xOffset_mm, int32_t yOffset_mm)

This command sets the antenna offset from CCRP. `xOffset_mm` indicates the offset starboard from CCRP to antenna in millimetres. `yOffset_mm` indicates the offset forward from CCRP to antenna in millimetres. If supported, both values will be saved by the radar and so do not need to be resent at each start up.

It will return "true" if the operation was successfully initiated, "false" otherwise.

4.1.37. bool SetAntennaType (uint16_t antennaType)

This command sets the antenna type. `antennaType` is the index to the antenna table from feature report. If supported, the antenna type will be saved by the radar and so do not need to be resent at each start up.

It will return "true" if the operation was successfully initiated, "false" otherwise.

4.1.38. bool SetToFactoryDefaults( )

This command resets the radar to the default values that were programmed in the factory.

It will return "true" if the operation was successfully initiated, "false" otherwise.

4.1.39. bool SetGuardZoneEnable(uint8_t zone, bool state)

This command enables or disables a guard zone where the radar can check if targets are entering or exiting a region. `zone` represents the guard zone number (0 or 1) and if `state` is "true" the zone will be enabled, if "false" it will be disabled.

It will return "true" if the operation was successfully initiated, "false" otherwise.

4.1.40. bool SetGuardZoneSetup(uint8_t zone, uint32_t startRange_m, uint32_t endRange_m, uint16_t bearing_deg, uint16_t width_deg)

This command sets up the sector for a specific guard zone. `zone` represents the guard zone number (0 or 1). `startRange_m` and `endRange_m` represent the beginning and end of the sector in metres. While `bearing_deg` represents the azimuth where the sector will be centred and `width_deg` represents the amplitude of the sector. A width of 360 degrees represents a circular sector.

It will return "true" if the operation was successfully initiated, "false" otherwise.

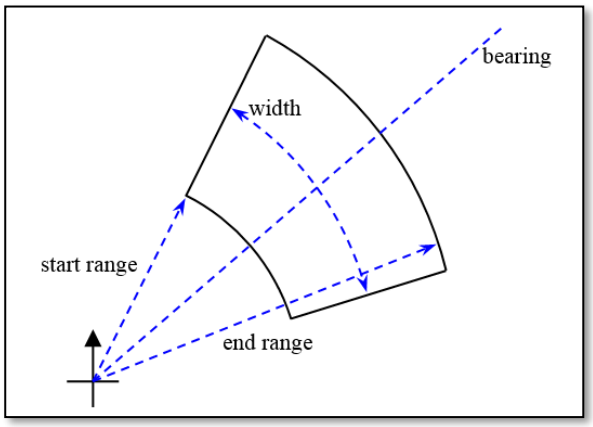

Figure 4-1: Guard zone definition.

4.1.41. bool SetGuardZoneAlarmSetup(uint8_t zone, eGuardZoneAlarmType type)

This command sets up the type of alarm we want to have generated for the specific guard zone. `zone` represents the guard zone number. The alarm `type` can be: `eGZAlarmEntry`, for targets entering in the sector; `eGZAlarmExit`, for targets exiting from the sector; and `eGZAlarmBoth`, for targets entering or exiting from the sector.

It will return "true" if the operation was successfully initiated, "false" otherwise.

4.1.42. bool SetGuardZoneAlarmCancel(uint8_t zone, eGuardZoneAlarmType type)

This command cancels an alarm that was raised by the radar from the guard zone module. `zone` represents the guard zone number that generated the alarm. The alarm `type` can be: `eGZAlarmEntry`, for a target that entered in the zone or `eGZAlarmExit` for a target that exited from the zone.

It will return "true" if the operation was successfully initiated, "false" otherwise.

4.1.43. bool SetGuardZoneSensitivity(uint8_t level)

This command sets the level of sensitivity for detecting targets entering or leaving any of the guard zones. `level` can be in the range 0-255, where 0 is minimum sensitivity and 255 is the maximum sensitivity.

It will return "true" if the operation was successfully initiated, "false" otherwise.

4.1.44. bool SetTuneState(bool automatic)

This command sets the tune between automatic and manual. If `automatic` is set to true, automatic tune will be used and the radar will start auto-tuning from the last manual value set.

It will return "true" if the operation was successfully initiated, "false" otherwise.

4.1.45. bool SetTuneCoarse(uint8_t level)

This command sets the coarse level of the manual tune.

It will return "true" if the operation was successfully initiated, "false" otherwise.

4.1.46. bool SetTuneFine(uint8_t level)

This command sets the fine level of the manual tune.

It will return "true" if the operation was successfully initiated, "false" otherwise.

4.2. Advanced commands to the radar

These commands all require a higher unlock level than those listed above before they can be used, and generally provide more precise control of radar features although at a lower level.

4.2.1. bool SetScannerRPM(unsigned rpm)

This command sets the speed of rotation for the scanner in revolutions-per-minute. The `rpm` parameter can assume values in the range 10 to 36 rpm inclusive, with ~24 rpm being the radars normal/standard speed.

Note that some functions, such as MARPA (target tracking), may be adversely affected at higer rotation speeds, especially at longer ranges. Also various factors can restrict the maximum RPM acheivable (see section 2.1 “RPM Limits”).

It will return “true” if the operation was successfully initiated, “false” otherwise.

4.2.2. bool SetUserMinSNR(float level_dB)

This command sets the Minimum Signal to Noise Ratio, which defines the lower limit of the combined STC curves. The `level_dB` can assume values in the range -100.0 to +100.0 inclusive.

It will return “true” if the operation was successfully initiated, “false” otherwise.

4.2.3. bool SetVideoAperture(float level_dB)

This command sets the STC Video-Aperture level, which deteremines the height of the vertical band that will be remapped/requantised into the final image samples. The `level_dB` can assume values in the range 0.0 to 100.0 inclusive.

It will return “true” if the operation was successfully initiated, “false” otherwise.

4.2.4. bool SetRangeSTCTrim(float level_dB)

This command sets the Range STC trimming value, which is an offset to the Range STC curves starting value. The `level_dB` can assume values in the range -100.0 to +100.0 inclusive.

It will return “true” if the operation was successfully initiated, “false” otherwise.

4.2.5. bool SetRangeSTCRate(float level_dBpDec)

This commands sets the Range STC rate value, which is the rate of fall for STC range curve. The `level_dBpDec` can be in the range of 0.1 to 10.0 dB per Decade inclusive.

It will return “true” if the operation was successfully initiated, “false” otherwise.

4.2.6. bool SetSeaSTCTrim(float level_dB)

This command sets the Sea STC trimming value, which is an offset to the Sea STC curves starting value. The `level_dB` can assume values in the range -100.0 to +100.0 inclusive.

It will return “true” if the operation was successfully initiated, “false” otherwise.

4.2.7. bool SetSeaSTCRate1(float level_dBpDec)

This commands sets the first Sea STC rate value, which is the rate of fall for the first part of the Sea STC curve. The `level_dBpDec` can be in the range of 0.1 to 10.0 dB per Decade inclusive.

It will return “true” if the operation was successfully initiated, “false” otherwise.

4.2.8. bool SetSeaSTCRate2(float level_dBpDec)

This commands sets the second Sea STC rate value, which is the rate of fall for second part of the Sea STC curve. The `level_dBpDec` can be in the range of 0.1 to 10.0 dB per Decade inclusive.

It will return “true” if the operation was successfully initiated, “false” otherwise.

4.2.9. bool SetRainSTCTrim(float level_dB)

This command sets the Rain STC trimming value, which is an offset to the Rain STC curves starting value. The `level_dB` can assume values in the range -100.0 to +100.0 inclusive.

It will return “true” if the operation was successfully initiated, “false” otherwise.

4.2.10. bool SetRainSTCRate(float level_dBpDec)

This commands sets the Rain STC rate value, which is the rate of fall for Rain STC curve. The `level_dBpDec` can be in the range of 0.1 to 10.0 dB per Decade inclusive.

It will return “true” if the operation was successfully initiated, “false” otherwise.

4.2.11. bool SetSectorBlanking (uint32_t sectorID, bool state)

This command sets the enabled state of the sector blanking. If `state` is set to true and the radar supports sector blanking of index `sectorID`, signal will not be transmitted within this sector.

It will return “true” if the operation was successfully initiated, “false” otherwise.

4.2.12. bool SetSectorBlankingSetup (uint32_t sectorID, bool state, uint16_t startBearing_degrees, uint16_t endBearing_degrees)

This command sets the parameters of sector blanking. If the radar supports sector blanking, the sector of index `sectorID` will be defined as the region between `startBearing_degrees` and `endBearing_degrees`, relative to the front of the radar. If `state` is set to true, signal will not be transmitted within this sector.

It will return “true” if the operation was successfully initiated, “false” otherwise.

4.2.13. bool SetMainBangSuppression (bool state)

This command sets the main bang suppression enabled state.

It will return “true” if the operation was successfully initiated, “false” otherwise.

4.3. Messages from the radar

This section describes the state messages a radar can send, and that the `tNRPImageClient` class will decode and pass on to all registered observer classes derived from `Navico::Protocol::NRP::iImageClientStateObserver`.

Note that the update call-back methods are called from a separate thread so you will need to ensure thread safe access to any data maintained external to the `tImageClient` object. Also certain methods, such as the ones used to add and remove observers should not be invoked from the call-backs as they may result in a deadlock.

4.3.1. void iImageClientStateObserver::UpdateMode(const tMode *pMode)

This message contains the operating mode of the radar. It is automatically sent regularly by the radar but you may also force the radar to send it immediately by using the `tImgeClient::QueryAll` or `tImgeClient::QueryMode` methods.

```
struct tMode
{
    uint32_t  state;
    uint32_t  ttState;
    uint32_t  warmupTime_sec;
    uint32_t  ttCount_sec;
};
```

Field	Description

state	One of the following values:
	`enum eScannerState`
	`{`
	`  eOff  = 0,`
	`  eStandby  = 1,`
	`  eTransmit  = 2,`
	`  eWarming  = 3,`
	`  eNoScanner  = 4,`
	`  eDetectingScanner = 5,`
	`};`

ttState	Timed Transmit state. The radar is in timed-transmit mode if
warmupTime_sec	The number of seconds needed to move from the *warming-up* state to the *stand-by* state
ttCount_sec	Time remaining before Timed Transmit mode will cause the radar to change state from *stand-by* to *transmit* or visa-versa

4.3.2. void iImageClientStateObserver::UpdateSetup(const tSetup *pSetup)

This message contains information about current setup of the radar. Current range selection, gains and guard zone settings are contained in this message. The `tImgeClient::QueryAll` or `tImgeClient::QuerySetup` methods may be used to explicitly ask the radar to send this information.

```
struct tGainControl
{
    uint32_t  type;                  // One of eUserGainManualAuto enum values
    uint8_t   value;                 // Manual gain (valid only if 'type' is eUserGainManual)
};

struct tFTCControl
{
    uint32_t  type;                  // unused
```

```
    uint8_t   value;                  // FTC level (0-255)
};

struct tGuardZone
{
    uint32_t  orientation;            // Relative to vessel (0) or north/absolute (1)
    uint32_t  rangeStart_m;           // Close range from boat (metres)
    uint32_t  rangeEnd_m;             // Far range from boat (metres)
    uint16_t  azimuth_ddeg;           // Starting angle (10ths of a degree relative to orientation)
    uint16_t  width_ddeg;             // Width angle (10ths of a degree – deci-degrees)
};

struct tGuardZoneAlarmSetup
{
    tGuardZoneAlarmType alarmType;    // One of eGuardZoneAlarmType enum values
    uint8_t   enabled;                // Alarm enabled state, enabled if non-zero
};

struct tGuardZones
{
    uint8_t   sensitivity;            // Sensitivity (low 0-255 high)
    uint8_t   active[cMaxGuardZones]; // True if corresponding GZ is enabled
    tGuardZone zone[cMaxGuardZones];
    tGuardZoneAlarmSetup alarmType[cMaxGuardZones];
};

struct tSetup
{
    uint32_t  range_dm;               // Currently selected range (in 10ths of a metre)
    uint8_t  reserved;                // unused
    uint8_t  useMode;                 // Use mode
    tGainControl gain[eTotalSetupGains]; // Indexed by eSetupGainControls
    tFTCControl ftc;                  // Fast-Time-Constant level
    uint32_t  tuneType;               // Pulse radar only
    uint8_t   coarseTune;             // Pulse radar only
    uint8_t   fineTune;               // Pulse radar only
    uint32_t  interferenceReject;     // Interference-Reject level (0-3)
    uint32_t  targetStretch;          // Target-Stretch level
    uint32_t  targetBoost;            // Target-Boost (0-2)
    tPulseWidthType  pwType;          // Pulse width length type enumeration
    uint32_t  pwLength_ns;            // Pulse width length in nanoseconds
    tGuardZones guardzones;           // Guard-Zone setup
};
```

Field	Description
range_dm	The selected range expressed in 10^{ths} of a meter (decimetres). Note that this range corresponds to the range last requested via the `SetRange` method, the range covered by spokes is unlikely to be the same as this, due to overscan and hardware limitations. Use the `GetSpokeRange_mm` function to calculate the actual range represented by a spoke.

useMode	The selected use mode. Only valid if the selected use mode is supported by the radar. The following use modes are currently defined:
	`enum eUseMode`
	`{`
	`eUseMode_Custom,`
	`eUseMode_Harbour,`
	`eUseMode_Offshore,`
	`eUseMode_Buoy,`
	`eUseMode_Weather,`
	`eUseMode_Bird,`
	`eUseMode_Netfinder,`
	`eUseMode_SaRT,`

	eUseMode_Doppler, eUseMode_RTE, eTotalUseModes };

gain[]	The state and level of various gains. The following gains (indexes) are currently defined:
	enum eSetupGainControls
	{
	eSetupGain,
	eSetupSea,
	eSetupRain,
	eTotalSetupGains
	};

type	One of:
	enum eUserGainManualAuto
	{
	eUserGainManual = 0x00,
	eUserGainAuto = 0x01,
	eUserGainAutoHarbour = eUserGainAuto,
	eUserGainAutoOffshore = 0x02,
	};

value	Only valid if manual gain has been selected. Assumes values in the range 0-255
ftc	tFTCControl structure containing Fast-Time-Constant formation. Currently only a single field is used
value	Assumes values in the range 0-255
tuneType coarseTune fineTune	Only used with PULSE radars - not used with BROADBAND or PULSE COMPRESSION radars
interferenceReject	The level of interference reject. Normally it assumes a value in the range 0-3
targetStretch	The amount to stretch targets. It assumes the value 1 if it is enabled, 0 if it is disabled
targetBoost	The amount of target-boost (emphasis of targets). It assumes values in the range 0-2
pwType pwLength_ns	Information about pulse width and pulse length for PULSE radar, or equivalent values for BROADBAND and PULSE COMPRESSION radars
gaurdzones	Information about the setup of guard zones. Currently only 2 guard zones are supported
sensitivity	The sensitivity of all guard zones. Assumes a level between 0-255 (0 least sensitive, 255 maximum sensitivity)
active[]	The enabled state of each of the guard-zones. Assumes 1 if enabled, 0 if disabled
zone[]	Contains the size and position setup information for each of the guard zones using the tGuardZone structure defined above. See 4.1.41 bool SetGuardZoneSetup(uint8_t zone, uint32_t startRange_m, uint32_t endRange_m, uint16_t bearing_deg, uint16_t width_deg) for a more complete explanation
orientation	Orientation of the guard-zone. 0 implies relative to the vessel, 1 relative to north
rangeStart_m	Distance from the radar to the start of the guard-zone sector, in metres
rangeEnd_m	Distance from the radar to the end of the guard-zone sector, in metres

azimuth_ddeg	Bearing for the centre of the guard-zone sector relative to the radars zero-bearing setting (boat head), in 10^{ths} of a degree (deci-degrees)
width_ddeg	Width of the guard-zone sector (centred on `azimuth`), in 10^{ths} of a degree. A width of 360 degrees (value 3600) indicates a guard-zone that completely encircles the boat
alarmType[]	Contains the alarm configuration of the guard zones using the `tGuardZoneAlarm` structure

alarmType	Defines when an alarm should be triggered. One of the following values:
	`enum eGuardZoneAlarmType`
	`{`
	`  eGZAlarmEntry = 0x0,`
	`  eGZAlarmExit  = 0x1,`
	`  eGZAlarmBoth  = 0x2,`
	`};`

enabled	Whether alarm generation is currently enabled

4.3.3. void iImageClientStateObserver::UpdateSetupExtended(const tSetupExtended *pSetupExtended)

This message contains additional information to the normal setup of the radar. The `tImgeClient::QueryAll` or `tImgeClient::QuerySetup` methods may be used to explicitly ask the radar to send this information.

```
struct tSetupExtended
{
    uint8_t  stcCurveType;            // Curve type for automatic sea clutter rejection
    uint8_t  localIR;                 // Local interference reject (off 0-3 high)
    uint8_t  fastScanMode;            // Faster scanner speed at shorter ranges
    tGainControl sidelobe;            // Sidelobe processing state
    uint16_t rpmX10;                  // Scanner rotation speed (10ths of RPM)
    uint8_t  noiseReject;             // Noise rejection (off 0-2 high)
    uint8_t  beamSharpening;          // Beam sharpening (off 0-3 high)
    tUserGain sea;                    // Sea gain with auto offset
    uint8_t  reserved[4];
};
```

Field	Description
stcCurveType	Contains the level of STC curve being applied to the automatic sea clutter rejection modes. It can assume values in the range 0-2, where 0 is for a calm sea, and 2 for rough sea
localIR	Local interference-reject setting, used to help reduce interference caused by other on-board devices. It can assume a value in the range 0-1
fastScanMode	Indicates that the radar is running in fast scan mode. This means the scanner will have a higher rotation rate at shorter ranges. It assumes the value 0 for normal/standard scanner rotation speed, and 1 for medium (36rpm) and 2 for high (48rpm)
sidelobe	The state and level of side-lobe processing

type	One of:
	`enum eUserGainManualAuto`
	`{`
	`  eUserGainManual  = 0x00,`
	`  eUserGainAuto  = 0x01,`
	`};`

value	Only valid if manual gain has been selected. Assumes values in the range 0-255

rpmX10	Contains the actual scanner RPM rate, in 10^{ths} of rotations per minute
noiseReject	Contains the level of noise-rejection being applied. Assumes values in the range 0-2, with 0 being off and 2 high
beamSharpening	Contains the level of beam-sharpening being applied. Assumes values in the range of 0-3, with 0 being off and 3 being high
sea	Contains the manual gain level and automatic gain offset. Assumes values in the range 0-255 for the manual gain level and -128 to 127 for the auto gain offset

4.3.4. void iImageClientStateObserver::UpdateProperties(const tProperties *pProperties)

This message contains information about the properties of the radar. The `tImgeClient::QueryAll` or `tImgeClient::QueryProperties` methods may be used to explicitly ask the radar to send this information.

```
struct tProperties
{
    uint16_t  gwRegVersionMinor;
    uint16_t  gwRegVersionMajor;
    uint16_t  gwVersionMinor;
    uint16_t  gwVersionMajor;
    uint16_t  gwCompileSource;
    uint16_t  scannerSwVersionBuild;
    uint8_t   supportsFeatures;
    uint8_t   _reserved[3];
    uint32_t  powerCycles;
    uint16_t  driverVersionMajor;
    uint16_t  driverVersionMinor;
    uint32_t  scannerId;
    tScannerType scannerType;
    uint32_t  operationTime_hour;
    uint32_t  warmupTime_sec;
    uint32_t  maxRange_dm;
    uint16_t  scannerSwVersionMajor;
    uint16_t  scannerSwVersionMinor;
    uint32_t  radarSwVersionMajor;
    uint32_t  radarSwVersionMinor;
    wchar_t   buildDate[16];
    wchar_t   buildTime[16];
    uint32_t  radarProtocolVersion;
    uint8_t   supportsScannerDetail;
    uint8_t   supportsMaxStopTxSectors;
    uint8_t   supportsParking;
};
```

Field	Description
powerCycles	Number of times the radar has been power-cycled. Only available in BR-4G and later radars

scannerType	The type of radar/scanner connected. One of the following values:
	`enum eScannerType`
	`{`
	`eNKE_1065 = 0,`
	`eNKE_249 = 1,`
	`eNKE_250_4 = 2,`
	`eNKE_250_4_NAX = 3,`
	`eNKE_2102_6 = 4,`
	`eNKE_2252_7 = 5,`

	eNKE_2252_9 = 6, e4kWSimulator = 7, eGWTestScanner = 8, eTypeNoScanner = 9, eFMCW400_BR24 = 10, eFMCW400_Simulator = 11, eFMCW400_HD3G = 12, eFMCW400_HD4G = 13, ePCMP_HALO = 14, ePROP_MAGGIE = 15, eTotalScannerTypes };
warmupTime_sec	Number of seconds necessary to perform a full warm-up
operationTime_hour	Total number of hours the radar has transmitted for
maxRange_dm	The maximum range the radar is capable of, expressed in 10^{ths} of a metre (decimetres)
radarSwVersionMajor radarSwVersionMinor buildDate buildTime	Information on the version of the software
. . .	All other parameters are of minimal importance

4.3.5. void iImageClientStateObserver::UpdateConfiguration(const tConfiguration *pConfiguration)

This message contains information about the configuration of the radar. The `tImgeClient::QueryAll` or `tImgeClient::QueryConfiguration` methods may be used to explicitly ask the radar to send this information.

```
struct tSigned24
{
    uint16_t lsw;                       // Least significant word
    int8_t   msb;                       // Most significant byte
int32_t Get() const { … }               // Get value as a 32-bit signed integer
void Set(int32_t value) { … }           // Set value to the 32-bit signed integer
};

struct tBlankSector
{
    Uint8_t   state;                    // Non-zero if this blank sector is enabled
    uint16_t  sectorStart_ddeg;         // Start of sector (0-360 degrees in 10ths of a degree)
    uint16_t  sectorEnd_ddeg;           // End of sector (0-360 degrees in deci-degrees)
};

struct tConfiguration
{
    uint32_t  zeroRange_mm;             // Pulse radars only
    uint16_t  zeroBearing_ddeg;         // Zero bearing offset (0-360 degrees in 10ths of a degree)
    uint16_t  parkPosition_ddeg;        // Parking angle of radar (0-360 degrees in deci-degrees)
    uint32_t  antennaHeight_mm;         // Height of the antenna above the water (milli-metres)
    uint32_t  cableLength;
    uint8_t   antennaType;
    uint8_t   ledLevel;
    tSigned24 antennaOffsetX_mm;
    tSigned24 antennaOffsetY_mm;
    uint32_t  timedTransmitPeriod_s;
    uint32_t  timedStandbyPeriod_s;
    tBlankSector blankSectors[cMaxBlankSectors];
    uint16_t  _3;                       // Reserverd
};
```

Field	Description
zeroRange_mm	Zero range information which only has meaning for PULSE radars
zeroBearing_ddeg	The zero bearing offset of the radar. A correction for installations where the zero-bearing position of the radar is not perfectly aligned with the head of the boat. It is expressed 10^{ths} of a degree, so it can assume values in the range 0-3599
parkPosition_ddeg	The parking angle of the radar. It is expressed 10^{ths} of a degree (deci-degrees), so it can assume values in the range 0-3599. This parameter is not used for BROADBAND radars
antennaHeight_mm	The height of the antenna, expressed in millimetres. Required for the correct operation of some of the automatic tuning modes, such as sea-clutter
ledLevel	The intensity level of LEDs
antennaOffsetX_mm	The offset starboard from CCRP to antenna in millimetres
antennaOffsetY_mm	The offset forward from CCRP to antenna in millimetres
blankSectors	The sector parameters for sector blanking
state	The enabled state of this sector. Non-zero if enabled
sectorStart_ddeg	The bearing of start of this blank sector. It is expressed 10^{ths} of a degree (deci-degrees), so it can assume values in the range 0-3599
sectorEnd_ddeg	The bearing of end of this blank sector. It is expressed 10^{ths} of a degree (deci-degrees), so it can assume values in the range 0-3599
. . .	All other parameters are of minimal importance

4.3.6. void iImageClientStateObserver::UpdateGuardZoneAlarm(const tGuardZoneAlarm *pAlarm)

This message is used to communicate to the client about guard-zone alarms.

```
struct tGuardZoneAlarm
{
    tGuardZoneSelect     zone;         // Number of guard-zone that raised the alarm
    tGuardZoneAlarmType  type;         // Type of alarm that has been raised
    tAlarmState          state;        // Current state of the raised alarm
};
```

Field	Description
zone	The guard zone number that raised the alarm

type	The type of alarm that has been raised. It can assume the following values:
	`enum eGuardZoneAlarmType`
	`{`
	`  eGZAlarmEntry = 0x0,`
	`  eGZAlarmExit = 0x1,`
	`  eGZAlarmBoth = 0x2,`
	`};`

state	The alarm state. It can assume the following values:
	`enum eAlarmState`
	`{`
	`  eAlarmActive = 0x1,`
	`  eAlarmInactive = 0x2,`

```
        eAlarmCancelled = 0x3,
    };
```

4.3.7. void iImageClientStateObserver::UpdateRadarError(const tRadarError *pError)

This message is used to report radar error conditions to the client.

```
struct tRadarError
{
    tRadarErrorType  type;              // One of eRadarErrorType enum values
    uint32_t  param1;                   // Information related to the error
    uint32_t  param2;                   // Further information related to the error
};
```

Field	Description

type	The type of error that occurred, one of the following enum values:
	`enum eRadarErrorType`
	`{`
	`eErrorPersistenceCorrupt = 0x00000001,`
	`eErrorZeroBearingFault = 0x00010001,`
	`eErrorBearingPulseFault = 0x00010002,`
	`eErrorMotorNotRunning = 0x00010003,`
	`eErrorCommsNotActive = 0x00010004,`
	`eErrorMagnetronHeaterVoltage = 0x00010005,`
	`eErrorModulationVoltage = 0x00010006,`
	`eErrorTriggerFault = 0x00010007,`
	`eErrorVideoFault = 0x00010008,`
	`eErrorFanFault = 0x00010009,`
	`eErrorScannerConfigFault = 0x0001000A,`
	`eErrorPowerSupplyTransient = 0x0001000B,`
	`eErrorScannerDetectFail = 0x0001000C,`
	`};`

param1 param2	Additional information related to the error condition raised

4.4. Advanced messages from the radar

These update call-backs all require a higher unlock level than those listed above before they can be used.

4.4.1. void iImageClientStateObserver::UpdateAdvancedState(const tAdvancedSTCState * pState)

This message is used to communicate with the client about advanced configuration options. The `tImgeClient::QueryAll` or `tImgeClient::QueryAdvancedSetup` methods may be used to explicitly ask the radar to send this information.

```
struct tAdvancedSTCState
{
    float  rangeSTCRef_dB;             // Range curve starting/reference value
    float  seaSTCRef_dB;;              // Sea curve starting/reference value
    float  rainSTCRef_dB;              // Rain curve starting/reference value

    float  rangeSTCTrim_dB;            // Range curve starting value adjustment
    float  seaSTCTrim_dB;              // Sea curve starting value adjustment
    float  rainSTCTrim_dB;             // Rain curve starting value adjustment

    float  rangeSTCRate_dBpDec;        // Range curve slope/rate-of-fall
    float  seaSTCRate1_dBpDec;         // Sea curve slope/rate-of-fall 1
    float  seaSTCRate2_dBpDec;         // Sea curve slope/rate-of-fall 2
    float  rainSTCRate_dBpDec;         // Rain curve slope/rate-of-fall

    float  userMinSNR_dB;              // Minimum Signal to Noise Ratio
    float  videoAperture_dB;           // Video aperture level
};
```

Field	Description
rangeSTCRef_dB seaSTCRef_dB rainSTCRef_dB	Reference/normal starting levels for STC range, sea, and rain curves respectively
rangeSTCTrim_dB seaSTCTrim_dB rainSTCTrim_dB	Trimming values for STC range, sea, and rain curves respectively. These values are added to the STC reference levels above to define the starting level of each STC curve
rangeSTCRate_dBpDec seaSTCRate1_dBpDec seaSTCRate2_dBpDec rainSTCRate_dBpDec	Rate of fall values for range, sea, and rain STC curves respectively. They can assume values in the range 0.1 to 10.0 dB per Decade
userMinSNR_dB	User defined minimum signal to noise ratio (lower limit for the STC curves). Assumes values in the range -100.0 to +100.0
videoAperture_dB	The current STC Video-Aperture level. It can assume values in the range 0.0 to 100.0

4.5. Features from the radar

This section describes the features reporting interface to a radar, and that the `tFeatureManager` class will decode and pass on to all registered observer classes derived from `Navico::Protocol::NRP::iFeatureObserver`.

4.5.1. void iFeatureObserver::UpdateFeature(const tFeatureEnum *pFeature)

The `pFeature` points to a `tFeatureEnum`, which refers to the exact feature that is changed in the radar. The detailed content of each feature is obtained from getter functions of `tFeatureManager`. Currently the following getter functions are available:

```
const tFeatureBase& GetFeatureCombinedNoiseIFReject();
const tFeatureSectorBlanking& GetFeatureSectorBlanking();
const tFeatureLevel& GetFeatureIR();
const tFeatureLevel& GetFeatureLocalIR();
const tFeatureLevel& GetFeatureNoiseReject();
const tFeatureLevel& GetFeatureRangeStretch();
const tFeatureLevel& GetFeatureTargetStretch();
const tFeatureLevel& GetFeatureBeamSharpening();
const tFeatureLevel& GetFeatureFastScan();
const tFeatureLevel& GetFeatureLED();
const tFeatureLevel& GetFeaturePerformanceMonitor();
const tFeatureLevel& GetFeatureStcCurves();
const tFeatureUseModes& GetFeatureSupportedUseModes();
const tFeatureRangeLimits& GetFeatureSidelobeGain();
const tFeatureRangeLimits& GetFeatureInstrRangeLimits();
const tFeatureGainLimits& GetFeatureSeaUserGainLimits();
const tFeatureAntennaTypes& GetFeatureSupportedAntennaTypes();
```

4.5.2. const tFeatureBase& tFeatureManager::GetFeatureCombinedNoiseIFReject()

This getter function returns the content to indicate the enabled and supported state of the combined noise IF reject feature.

```
struct tFeatureBase
{
    tFeatureEnum featureEnum; ;        // The type of feature
    bool         enabled;              // Whether the feature is enabled
    bool         supported;            // Whether the feature is supported
};
```

Field	Description
featureEnum	The feature that following content describes about
enabled	Whether this feature is currently enabled
supported	Whether this feature is supported by the radar

4.5.3. const tFeatureSectorBlanking& tFeatureManager::GetFeatureSectorBlanking()

This getter function returns the content to indicate the enabled, supported state and number of sectors supported of the sector blanking feature.

```
struct tFeatureSectorBlanking : tFeatureBase
{
    uint8_t sectorCount;               // The number of blank sectors supported by the radar
};
```

Field	Description
featureEnum	The feature that following content describes about (inherited from `tFeatureBase`)
enabled	Whether this feature is currently enabled (inherited from `tFeatureBase`).
supported	Whether this feature is supported by the radar (inherited from `tFeatureBase`)
sectorCount	The number of blank sectors supported by the radar

4.5.4. const tFeatureLevel& tFeatureManager::GetFeatureIR()

This getter function returns the content to indicate the enabled, supported state and maximum level supported of the IR feature.

```
struct tFeatureLevel : tFeatureBase
{
uint8_t maxLevel;                    // The maximum level supported by the radar
uint32_t mask;                       // Reserved
};
```

Field	Description
featureEnum	The feature that following content describes about (inherited from `tFeatureBase`)
enabled	Whether this feature is currently enabled (inherited from `tFeatureBase`)
supported	Whether this feature is supported by the radar (inherited from `tFeatureBase`)
maxLevel	The maximum level supported by the radar

4.5.5. const tFeatureLevel& tFeatureManager::GetFeatureLocalIR()

This getter function returns the content to indicate the enabled, supported state and maximum level supported of the local IR feature.

4.5.6. const tFeatureLevel& tFeatureManager::GetFeatureNoiseReject()

This getter function returns the content to indicate the enabled, supported state and maximum level supported of the noise reject feature.

4.5.7. const tFeatureLevel& tFeatureManager::GetFeatureRangeStretch()

This getter function returns the content to indicate the enabled, supported state and maximum level supported of the range stretch feature.

4.5.8. const tFeatureLevel& tFeatureManager::GetFeatureTargetStretch()

This getter function returns the content to indicate the enabled, supported state and maximum level supported of the target stretch feature.

4.5.9. const tFeatureLevel& tFeatureManager::GetFeatureBeamSharpening()

This getter function returns the content to indicate the enabled, supported state and maximum level supported of the beam sharpening feature.

4.5.10. const tFeatureLevel& tFeatureManager::GetFeatureFastScan()

This getter function returns the content to indicate the enabled, supported state and maximum level supported of the fast scan feature.

4.5.11. const tFeatureLevel& tFeatureManager::GetFeatureLED()

This getter function returns the content to indicate the enabled, supported state and maximum level supported of the LED feature.

4.5.12. const tFeatureLevel& tFeatureManager::GetFeaturePerformanceMonitor()

This getter function returns the content to indicate the enabled, supported state and maximum level supported of the performance monitor feature.

4.5.13. const tFeatureLevel& tFeatureManager::GetFeatureStcCurves()

This getter function returns the content to indicate the enabled, supported state and maximum level supported of the STC curves feature.

4.5.14. const tFeatureUseModes& tFeatureManager::GetFeatureSupportedUseModes()

This getter function returns the content to indicate the enabled, supported state and array of use modes of the supported use modes feature.

```
struct tFeatureUseModes : tFeatureBase
{
uint8_t useModeCount;                          // The number of use modes in the array
eUseMode useModes[cUseMode_Total];             // Array of supported use modes
};
```

Field	Description
featureEnum	The feature that following content describes about (inherited from `tFeatureBase`)
enabled	Whether this feature is currently enabled (inherited from `tFeatureBase`)
supported	Whether this feature is supported by the radar (inherited from `tFeatureBase`)
useModeCount	The number of use modes in the array
useModes	The array of use mode enumerators that are supported by the radar

4.5.15. const tFeatureRangeLimits& tFeatureManager::GetFeatureSidelobeGain()

This getter function returns the content to indicate the enabled, supported state, minimum and maximum value of the side lobe gain feature.

```
struct tFeatureRangeLimits : tFeatureBase
{
uint32_t minimum;
```

```
uint32_t maximum;
};
```

Field	Description
featureEnum	The feature that following content describes about (inherited from `tFeatureBase`)
enabled	Whether this feature is currently enabled (inherited from `tFeatureBase`)
supported	Whether this feature is supported by the radar (inherited from `tFeatureBase`)
miniumum	The minimum value that can be accepted by the radar
maximum	The maximum value that can be accepted by the radar

4.5.16. const tFeatureRangeLimits& tFeatureManager::GetFeatureInstrRangeLimits()

This getter function returns the content to indicate the enabled, supported state, minimum and maximum value of the instrument range limits feature.

4.5.17. const tFeatureGainLimits& tFeatureManager::GetFeatureSeaUserGainLimits()

This getter function returns the content to indicate the enabled, supported state, minimum and maximum of manual level, and minimum and maximum of auto offset of the sea user gain feature.

```
struct tFeatureGainLimits : tFeatureBase
{
uint8_t manualLevelMin;
uint8_t manualLevelMax;
int8_t autoOffsetMin;
int8_t autoOffsetMax;
};
```

Field	Description
featureEnum	The feature that following content describes about (inherited from `tFeatureBase`)
enabled	Whether this feature is currently enabled (inherited from `tFeatureBase`)
supported	Whether this feature is supported by the radar (inherited from `tFeatureBase`)
manualLevelMin	The minimum of manual level that can be accepted by the radar
manualLevelMax	The maximum of manual level that can be accepted by the radar
autoOffsetMin	The minimum of auto offset that can be accepted by the radar
autoOffsetMax	The maximum of auto offset that can be accepted by the radar

4.5.18. const tFeatureAntennaTypes& tFeatureManager::GetFeatureSupportedAntennaTypes()

This getter function returns the list of antenna types supported by the radar.

```
struct tAntennaType
{
uint16_t size;
};
```

```
struct tFeatureAntennaTypes : tFeatureBase
{
static const int cAntennaType_Total = 8;

uint8_t count;
uint8_t antennaTypes [cAntennaType_Total];
};
```

Field	Description
featureEnum	The feature that following content describes about (inherited from `tFeatureBase`)
enabled	Whether this feature is currently enabled (inherited from `tFeatureBase`)
supported	Whether this feature is supported by the radar (inherited from `tFeatureBase`)
count	The number of antenna types supported by the radar
antennaTypes	The antenna types supported by the radar

4.6. Spokes from the radar

This section describes the image/spoke messages a radar sends, and that the `tNRPImageClient` class will decode and pass on to all registered observer classes derived from `Navico::Protocol::NRP::iImageClientSpokeObserver`.

4.6.1. void iImageClientSpokeObserver::UpdateSpoke(const Spoke::t9174Spoke *pSpoke)

The `pSpoke` contains all the information to build the radar image. See section 3.1 for more information about spoke definitions.

4.7. Class initialisation and configuration

This describes other methods used in `tImageClient` class to initialise and configure it properly.

4.7.1. int Connect(const char * pSerialNumber, unsigned imageStream)

This method connects a `tImageClient` object to a specific radar, opening multicast sockets and starting its internal threads so that it can send and receive messages to and from the radar.

The parameter `pSerialNumber` must point to the valid serial-number of a radar on the network which supports the `imageStream` number provided. A list of radars attached to the network, and how many image-streams they support, can be obtained from the `tMultiRadarClient` class.

It will return 0 if the operation was successful, or one of the `Protocol::eError` enum values otherwise.

4.7.2. bool Disconnect()

This method disconnects the `tImageClient` object from the network, closing all open network sockets and shutting down its internal threads.

It will return “true” if the operation was successful, “false” otherwise.

4.7.3. bool AddStateObserver(iImageClientStateObserver *pObserver)

This method adds an observer to all “state” messages that can come from the radar. Multiple observers can be registered, and each of them will be informed on the reception of a state message using the appropriate call-back.

Note that the update call-back methods are called from a separate thread so you will need to ensure thread safe access to any data maintained external to the `tImageClient` object. Also certain methods, such as this one should not be invoked from the call-backs as they may result in a deadlock.

It will return “true” if the operation was successful, “false” otherwise.

4.7.4. bool RemoveStateObserver(iImageClientStateObserver *pObserver)

This method removes a previously registered state observer. After an observer has been removed, it will not receive any further updates.

It will return “true” if the operation was successful, “false” otherwise.

4.7.5. bool AddSpokeObserver(iImageClientSpokeObserver *pObserver)

This method adds an observer to the image data sent by the radar. Multiple observers can be registered, and each of them will be informed upon the reception of a spoke image data.

Note that the update call-back methods are called from a separate thread so you will need to ensure thread safe access to any data maintained external to the `tImageClient` object. Also certain methods, such as this one should not be invoked from the call-backs as they may result in a deadlock.

It will return “true” if the operation was successful, “false” otherwise.

4.7.6. bool RemoveSpokeObserver(iImageClientSpokeObserver *pObserver)

This method removes a previously registered spoke observer. After an observer has been removed, it will not receive any further updates.

It will return “true” if the operation was successful, “false” otherwise.

4.7.7. void GetVersion(uint32_t &major, uint32_t &minor, uint32_t &build)

This method returns the class version. Usually the version is indicated as `major.minor.build`.