Technical Paper

NavRadar – BRPC SDK

Multi-Radar Client class description

Document Version:

D [Release], Date: 05-Sep-2016

Abstract:

This document describes how to use Multi-Radar Client Class to interface software with Navico Radars.

Document:	Total:
2	**4**

Revision History:

A	9-Dec-2009	Document created	Graeme Bell
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	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
Device-ID	A radar/device identifier that determines a radars set of communication addresses.
Serial-Number	Radars unique ASCII serial number
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 `NRP Multi-Radar` class to ascertain what radars are available on a network, and unlock them ready for use with the Image and Target Tracking classes.

2 Multi-Radar

The communications protocol is based on multicast UDP/IP, with each radar using the same set of multicast addresses for sending and receiving identification, negotiation, and general configuration messages.

2.1 Radar Device ID's

Every radar on a network has a device-ID that establishes which set of network addresses it will communicate over. These are negotiated when a radar is first connected to a network, or restarted.

Older radars have a standard fixed device-ID so only one of these can exist on a network at any time. Newer radars will use this standard device-ID if they are they only radar on the network, otherwise they will negotiate a new device-ID that doesn't conflict with any other radar on the network.

Normally you do not need to be concerned about a radars device-ID, in fact they are not even exposed externally from the library, but under some circumstances you may want to force a radar to renegotiate its device-ID (via the tMultiRadarClient::ResetDeviceIDs method). Such as when a radar has previously been on a network with others (so is not using the standard device-ID) but is now going to be the only radar on the network.

2.2 Unlocking Radars

By default a radar is locked and needs to be unlocked before the other classes in this library can interact with it. Multiple unlock levels exist and will unlock increasing radar/library capabilities; currently just three levels are supported.

Level	Description
0	The radar and libraries are locked. Only the `tMultiRadarClient` and `tPPI` classes can be used. The `tImageClient` and `tTargetTrackingClient` classes will not operate.
1	All classes can be used, but only with a radar that has the standard device-ID (only one radar on the network). Also the advanced `tImageClient` functionality is not available.
2	All classes can be used; even if the radar does not have the standard device-ID (ie. multi-radar support is enabled). All methods, including `tImageClient` advanced functionality is available.

Table 2-1: Unlock Levels

To unlock a radar you must first acquire an unlock-key from Navico. For this you will need to supply the radars serial-number, and its lock-ID. These can both be found by using the example GUIDemo application provided with the library (try selecting and unlocking a radar in the Multi-Radar section of the application).

Once you have an unlock-key for a radar you can use the `tMultiRadarClient::UnlockRadar` method to unlock it. You may provide the unlock-key to the library in one of several ways. Either by telling the library about it before hand, using the `tMultiRadarClient::SetUnlockKey` method. By providing an `iUnlockKeySupplier` object that can supply the unlock-key on demand, or directly in the second version of the `tMultiRadarClient::UnlockRadar` method.

3 Class `tMultiRadarClient`: definition and usage

The `tMultiRadarClient` class is able to monitor and unlock radars without the client needing to worry about the network protocol. This isolates the clients from future changes to the multi-radar protocol.

This class works with the observer paradigm that means the object interested in receiving updates to the list of available radars and/or their lock-state needs to derive from the observer classes (`iRadarListObserver` and `iUnlockStateObserver`) and be registered with the `tMultiRadarClient` class.

One other call-back can be provided, in this case for providing unlock-keys to the library on an on-demand basis, for this the `iUnlockKeySupplier` needs to be implemented.

example:

```
#include <MultiRadarClient.h>

using Navico::Protocol;

class MyRadarClient
    : public iRadarListObserver
    , public iUnlockStateObserver
    , public iUnlockKeySupplier
{
...

public:
    // iRadarListObserver callback
    void UpdateRadarList( const char * pSerialNumber, eAction action );

    // iUnlockStateObserver callback
    void UpdateUnlockState( const char * pSerialNumber, int lockState );

    // iUnlockKeySupplier callback
    int GetUnlockKey( const char * pSerialNumber,
        const uint8_t * pLockID, unsigned lockIDSize,
        uint8_t * pUnlockKey, unsigned maxUnlockKeySize );
...
};
```

Registration of observers with the `tMultiRadarClient` class can be done using the two methods

```
bool AddRadarListObserver( iRadarListObserver *pObserver )
bool AddUnlockStateObserver( iUnlockStateObserver *pObserver ).
```

And the supplier with

```
void SetUnlockKeySupplier( iUnlockKeySupplier * pSupplier )
```

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

```
int tMultiRadarClient::Connect()
```

Once you have successfully connected the `tMultiRadarClient` will be able to transmit and receive commands to and from any radars. The reception of information will start immediately so it is possible for one of the call-backs to be invoked even before the `Connect` method has returned.

3.1 Methods

Below are details of the methods provided by the `tMultiRadarClient` for interacting with radars.

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.

3.1.1 int SetUnlockKey(const char * pSerialNumber, const uint8_t * pUnlockKey, unsigned unlockKeySize)

This method pre-defines an unlock-key for the specified radar and immediately initiates an unlock sequence (if `pUnlockKey` is not null). As an alternative you can provide the unlock-key on demand via the `iUnlockKeySupplier::GetUnlockKey` call-back.

The `pSerialNumber` parameter must point to the null-terminated ASCII string which uniquely identifies the radar, while the `pUnlockKey` parameter must point to a buffer containing the radars unlock-key of size `unlockKeySize`. If `pUnlockKey` is null any unlock-key being held for that radar will be cleared.

If the initiated unlock sequence is successful the `UpdateUnlockState` call-back will be invoked on all registered `iUnlockStateObservers`.

3.1.2 void ClearUnlockKeys()

This method clears all cached unlock-keys being held by the `tMultiRadarClient`.

3.1.3 int UnlockRadar(const char * pSerialNumber, uint32_t wait_ms = 0)

This method initiates an unlock sequence, waiting a specified maximum amount of time for it to complete. It assumes that the unlock-key has already been supplied (`tMultiRadarClient::SetUnlockKey`) or will be provided by an `iUnlockKeySupplier`.

The `pSerialNumber` parameter must point to the null-terminated ASCII string which uniquely identifies the radar, while the `wait_ms` parameter specifies the number of milliseconds to wait for the unlock sequence to complete. A value of 0 causes the method to return immediately after initiating an unlock sequence.

This method supports a special case for initiating unlocks sequences for all radars attached to the network. If `pSerialNumber` is null and `wait_ms` is 0 then this method will broadcast a request to all radars for their lock-ID's and return `EPending`.

Note care should be taken to ensure that any call-backs from the library to your application will not cause a deadlock while your application thread that is making this call is blocked.

3.1.4 int UnlockRadar(const char * pSerialNumber, const uint8_t * pUnlockKey, unsigned unlockKeySize, uint32_t wait_ms = 0)

This method initiates an unlock sequence, waiting a specified maximum amount of time for it to complete.

The `pSerialNumber` parameter must point to the null-terminated ASCII string which uniquely identifies the radar, while the `pUnlockKey` parameter must point to a buffer containing the radars unlock-key of size `unlockKeySize`. The `wait_ms` parameter specifies the number of milliseconds to wait for the unlock sequence to complete. A value of 0 causes the method to return immediately after initiating an unlock sequence.

Note care should be taken to ensure that any call-backs from the library to your application will not cause a deadlock while your application thread that is making this call is blocked.

3.1.5 unsigned GetRadars(char radars[][MAX_SERIALNUMBER_SIZE], unsigned maxRadars)

This method returns a list of serial-numbers for radars that have announced themselves on the network. The `radars` array provided must be able to hold at least `maxRadars` number of serial-numbers. The serial-numbers are returned as a simple null-terminated string of ASCII characters.

Although it will never put more than `maxRadars` number of serial-numbers in the `radars` array it returns the actual number of radars available, which may be larger than the `maxRadars` value.

3.1.6 int GetLockID (char lockID[MAX_LOCKID_SIZE *2], const char* pSerialNumber)

This method returns the lock ID of the radar with the specified serial number on the network in ASCII characters. If such radar is found, the lockID byte array will be used to hold the lock ID and the length of lock ID in number of bytes will be returned. If a radar cannot be found on the network to match the specified serial number, 0 will be returned. The lockID byte array must have size of at least MAX_LOCKID_SIZE * 2 to prevent buffer overflow.

3.1.7 void ClearRadars()

This method clears the list of radars currently held by the `tMultiRadarClient`. It will cause the `iRadarListObserver::UpdateRadarList` method to be invoked with an `action` value of `iRadarListObserver::Removed` for each radar currently in the list.

You will need to call the `tMultiRadar::QueryRadars` method to get the radars to re-announce themselves and hence repopulate the list.

3.1.8 bool QueryRadars()

This broadcasts a query command telling all radars to re-announce themselves on the network. Any radars not already in the radar list (see section 3.1.5 `tMultiRadarClient::GetRadarList`) will cause the `iRadarListObserver::UpdateRadarList` method to be invoked when they do their announcement.

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

3.1.9 bool ResetDeviceIDs()

This broadcasts a command telling all radars to reset their device-ID, forcing the radars to re-negotiate their device-id the next time they connect to the network (after being restarted or physically disconnected & reconnected).

A radars device-ID determines its network address, so this command may eventually result in existing radars changing their network address. Meaning any other clients connected to the old address will need to be `Disconnect`ed and re-`Connect`ed before they will operate correctly.

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

3.2 Available Radars - iRadarListObserver

Here are listed the `Navico::Protocol::iRadarListObserver` observer methods invoked when: a new radar is detected; an existing radar changes its address; or a radar is removed from the list of available ones.

This class lets you observe changes as they happen, but you may also get a list of available radars at any time by using `the tMultiRadarClient::GetRadars` method. While the `tMultiRadarClient::SendQuery` method may be used to force all radars to re-announce themselves to ensure the list is up to date.

3.2.1 void UpdateRadarList(const char * pSerialNumber, eAction action)

This call-back reports changes to the list of available radars. The `pSerialNumber` parameter points to a nul-terminated ascii string which uniquely identifies the radar. While `action` indicates what action has just taken place, and is one of,

eAction Values	Description
`Added`	A new radar has been added to the list

`Removed`	An existing radar has been removed from the list. Currently this only happens when the `tMultiRadarClient::ClearRadars` method is called.
`Changed`	The address of an existing radar has changed, so any Image or Target Tracking clients currently connected to it may need to be disconnected and reconnected.

3.3 Radars Unlocked States - iUnlockStateObserver

Here are listed the `Navico::Protocol::iUnlockStateObserver` observer methods invoked when the unlocked state of a radar changes.

3.3.1 void UpdateUnlockState(const char * pSerialNumber, int lockState)

This call-back reports changes to the unlocked state of a radar. The `pSerialNumber` parameter points to a nul-terminated ascii string which uniquely identifies the radar, while `lockState` indicates the level at which the radar has been unlocked (higher numbers indicate more features have been unlocked).

3.4 Supplying Unlock Keys - iUnlockKeySupplier

Here are listed the `Navico::Protocol::iUnlockKeySupplier` methods invoked when `tMultiRadarClient` is in the process of unlocking a radar and doesn't yet have an unlock key for a radar. Unlock keys may also be pre-provided by using the `tMultiRadarClient::SetUnlockKey` method.

3.4.1 int GetUnlockKey(const char * pSerialNumber, const uint8_t * pLockID, unsigned lockIDSize, uint8_t * pUnlockKey, unsigned maxUnlockKeySize)

This call-back requests an unlock key for a particular radar. The `pSerialNumber` parameter points to a nul-terminated ascii string which uniquely identifies the radar, while `pLockID` and `lockIDSize` provide the lock-ID for that radar.

The `pUnlockKey` parameter points to a buffer that is at least `maxUnlockKeySize` bytes in size, and into which you need to place the radars unlock-key. The value returned should be the actual size of the unock-key placed in the buffer, or -1 if you could not provide an unlock-key.

Any unlock-key provided will be cached and automatically reused (without this method being called) until either the `tMultiRadarClient::ClearUnlockKeys` method is called, or `tMultiRadarClient` is disconnected.

3.5 Class initialisation and configuration

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

3.5.1 static tMultiRadarClient * GetInstance( )

This method provides access to the single global instance of `tMultiRadarClient`. All other methods can only be applied to this object.

3.5.2 int Connect( )

This method connects the `tMultiRadarClient` object to the network, opening multicast sockets and starting its internal threads so that it can send and receive message to and from any radars.

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

3.5.3 bool Disconnect()

This method disconnects the `tMultiRadarClient` 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.

3.5.4 bool AddRadarListObserver(iRadarListObserver *pObserver)

This method adds an observer for all changes to the list of Radars held by the `tMultiRadarClient` object. Multiple observers can be registered, and each of them will be informed when a Radar is added, removed, or updated (when its Device-ID changes) from the list using the appropriate call-back.

Note that 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 `tMultiRadarClient` 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.

3.5.5 bool RemoveRadarListObserver(iRadarListObserver *pObserver)

This method removes a previously registered Radar List 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.

3.5.6 bool AddUnlockStateObserver(iUnlockStateObserver *pObserver)

This method adds an observer to changes in a radars locked state. Multiple observers can be registered, and each of them will be informed on lock state changes using the appropriate call-back.

Note that 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 `tMultiRadarClient` 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.

3.5.7 bool RemoveUnlockStateObserver(iUnlockStateObserver *pObserver)

This method removes a previously registered target 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.

3.5.8 void SetUnlockKeySupplier(iUnlockKeySupplier *pSupplier)

This method sets up a call-back for providing unlock-keys to the `tMultiRadarClient` class when it is in the process of unlocking a radar which it has not already been provided an unlock-key for. Only a single supplier can be setup, so any subsequent calls will just overwrite the previous supplier.

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