n-Core® API 2.0 User Manual

Rev. 20130823

Table of Contents

1 Introduction

n-Core® is a powerful hardware & software platform to develop, integrate and deploy easily and quickly, a wide variety of applications over Wireless Sensor Networks based on the IEEE 802.15.4/ZigBee™ standard.

The main features of the n-Core® platform are its fast deployment and ease of use because the hardware layer (i.e., n-Core® Sirius devices) are pre-loaded with a specific firmware, whose functionalities can be accessed from any PC via an **Application Programming Interface (API)** also developed by Nebusens (accessible from different development platforms like C/C++, .Net or Java, among many others, and **existing versions for Windows and Linux**), **without writing any line of embedded code**, which greatly facilitates its configuration and deployment.

The API includes two engines: an **automation engine** that can read virtually any sensor on the market (e.g., temperature, presence, lighting, etc.) and act on a wide range of actuators (e.g., alarms, sirens, electronic locks, etc.); and a **locating engine**, which allows determining the position of mobile nodes using the same network infrastructure.

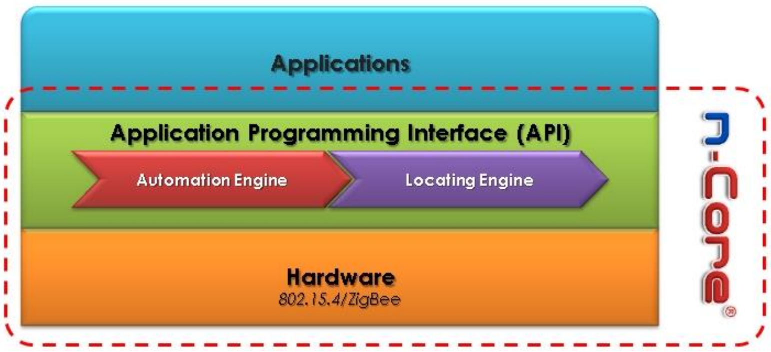


Figure 1. General schema of the n-Core® platform.

Summing up, the n-Core® platform is composed by a range of n-Core® Sirius devices (*hardware*), the *firmware* embedded on each device, a set of Application Programming Interfaces (API) and several configuration tools (e.g., the *n-Core® Configuration Tool*).

This document is centered on the application programming interfaces (API).

2 The n-Core® API

The n-Core® API allows developing end-user applications from almost any IDE in a simple way. The API is made up of a collection of several dynamic-link libraries (i.e., "*.dll*" files in the Windows version and "*.so*" files in the Linux implementation) that provides access to all the functionalities of the platform. This API offers two engines that make easier the development of specific applications:

1. **Automation Engine.** It allows controlling and monitoring sensors or actuators connected to n-Core® Sirius devices. It also offers functions that cover from the network creation to automatic data collection.
   - The Automation Engine can run directly over the n-Core Sirius devices. That is, there is no need to run any application on a PC.

2. **Locating Engine.** It offers several functions to develop *Real Time Location Systems* (RTLS). It is composed of different algorithms (*signpost* and *fuzzy*) that allows calculating the position of mobile n-Core® Sirius devices both indoors and outdoors.

There are **three different APIs for Windows and Linux systems**:

1. **n-Core++ (*ncore++*):** it is the main and native API of n-Core®. It manages all the communication between the PC and the n-Core® Sirius devices. All the functionalities are offered in C++ language.
2. **n-Core.C (*ncore.c*):** It is a wrapper of n-Core++ and offers all the functionalities in C language, easily linked from other languages such as Java or Python.
3. **n-Core.NET (*ncore.net*, only available for Windows):** It is a wrapper of n-Core.C and offers all the functionalities in .NET C# language.

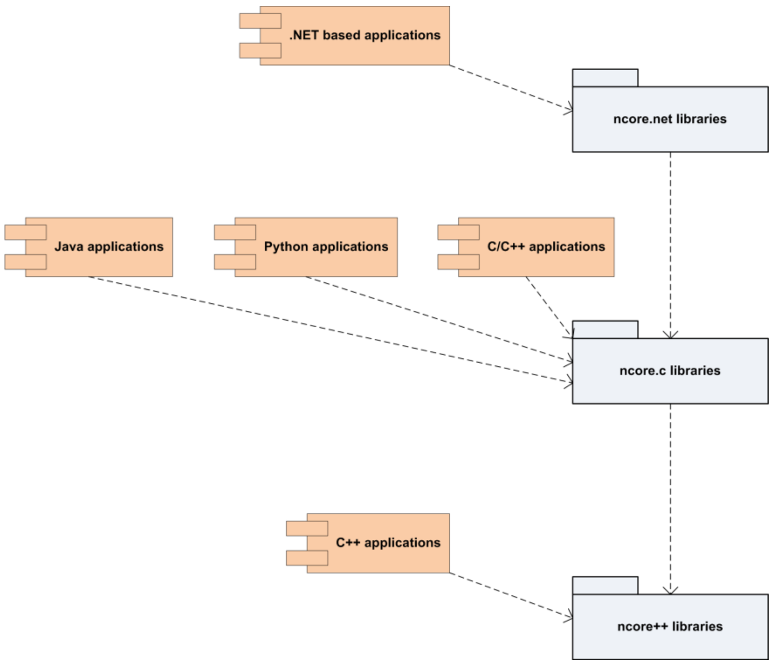


Figure 2. The three different n-Core® APIs.

2.1 The n-Core® Communication Protocol

n-Core® uses a proprietary communication protocol developed by Nebusens.

This protocol is used by the n-Core® Sirius devices to communicate between them. The n-Core® API translates the communication protocol and offers all the functionalities directly to the end-user applications.

There are **two main frames** used by the communication protocol:

1. **Command request/response.** These frames are used when a node (i.e., a n-Core® Sirius device) requests specific data from another node in the network. The first node (transmitter) sends a *command request* to the second node (receiver). Then, the receiver sends the transmitter a *command response* with the requested status and data.
   - End-user applications can send *command request* frames invoking methods or functions belonging to one of the *functional blocks* of the n-Core® API. For example:
     - Request information about a node (e.g., network ID, transmission power, etc.).
     - Request a node to send one of its input values periodically (e.g., send data from ADC, GPI or I^{2}C). The node responses with "*status ok*" and then sends further data using message frames periodically.

2. **Message.** These frames are used to send data from one node to another with no expected response by the receiver. For example:
   - Send the temperature from a sensor connected to a n-Core® Sirius device.
   - Exchange data between nodes or act as network interfaces between two computers.

2.2 Functional Blocks and the three n-Core® APIs

The n-Core® API is divided into several functional blocks sorted by the type of application. The API has different groups of classes, methods, functions and dynamic link libraries (dll) for every one of the functional blocks.

Therefore, the API of n-Core® is divided in a group of libraries with common functionalities (called *common*, *ports*, *frames*, *events*) and another group of libraries with specific functionalities:

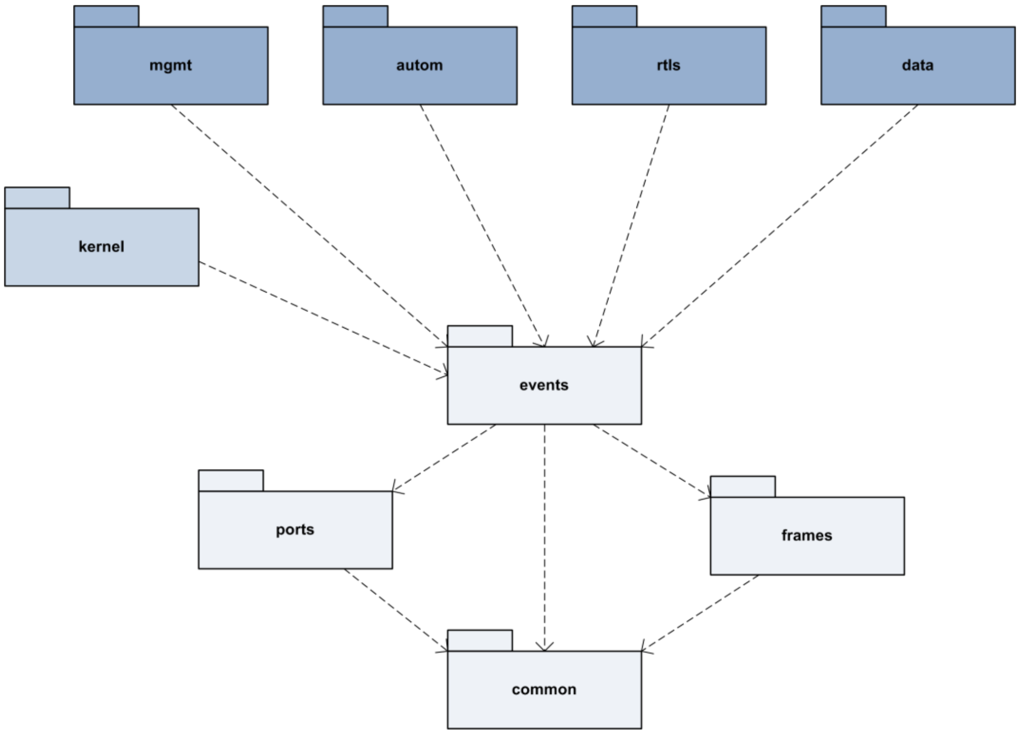


Figure 3. Functional blocks of the n-Core® APIs.

- **Common (*common*).** This functional block includes common features to all the rest of blocks, such as basic types, classes representing the nodes in the network, classes for representing parameters that can be modified in the nodes, or logging functions.
- **Ports (*ports*)**. It provides an abstraction of possible ports that can be used to exchange data between the API and devices attached to a n-Core® network, such as serial communication ports or socket ports.
- **Frames (*frames*).** This functional block provides the communication protocol used in n-Core®, as well as the different frames exchanged using it: command requests/responses and messages.
- **Events (*events*).** This functional block provides the core of the operation of the n-Core® API, including initialization functions, proxy classes to manage communication ports using the n-Core® protocol, as well as event handler and dispatcher classes to manage events coming from both the devices in the network and the locating algorithms in the *rtls* block.
- **Kernel (*kernel*).** This block provides some basic functionalities of the platform that cannot be included inside other functional block because of its nuclear behavior.
- **Management (*mgmt*).** It provides several basic functionalities to manage the wireless network. This block allows the user to modify the parameters of the ZigBee™ network (PAN Id, channel, node transmission power, etc.) or to discover the network topology (neighbor nodes, etc).
- **Automation (*autom*).** It provides the functionalities to manage the different sensors and actuators attached to the n-Core® *Sirius* devices. This block allows the user to access to the automation engine of the platform.

- **Location or RTLS (*rtls*).** It provides access to the location engine of the n-Core® platform, so it allows using the different location algorithms implemented in the platform.
- **Data (*data*).** This block provides the functionalities to manage the transmission of general data messages between the n-Core® *Sirius* devices. These messages can be used to connect different computers among them through the ZigBee™ network.

The different functional blocks are deployed in different dynamic link libraries according to the n-Core® API chosen (ncore++, ncore.c or ncore.net).

2.2.1 n-Core++ API (ncore++)

n-Core++ is the native n-Core® API, written in C++, and is formed by eight dynamic link libraries (i.e., "*.dll*" files in Windows and "*.so*" files in Linux) implementing the nine functional blocks:

- **ncore++_common.dll (libncore++_common.so)**
  - common
- **ncore++_ports.dll (libncore++_ports.so)**
  - ports
- **ncore++_frames.dll (libncore++_frames.so)**
  - frames
- **ncore++_events.dll (libncore++_events.so):**
  - events, kernel
- **ncore++_mgmt.dll (libncore++_mgmt.so)**
  - mgmt
- **ncore++_autom.dll (libncore++_autom.so)**
  - autom
- **ncore++_rtls.dll (libncore++_rtls.so)**
  - rtls
- **ncore++_data.dll (libncore++_data.so)**
  - data

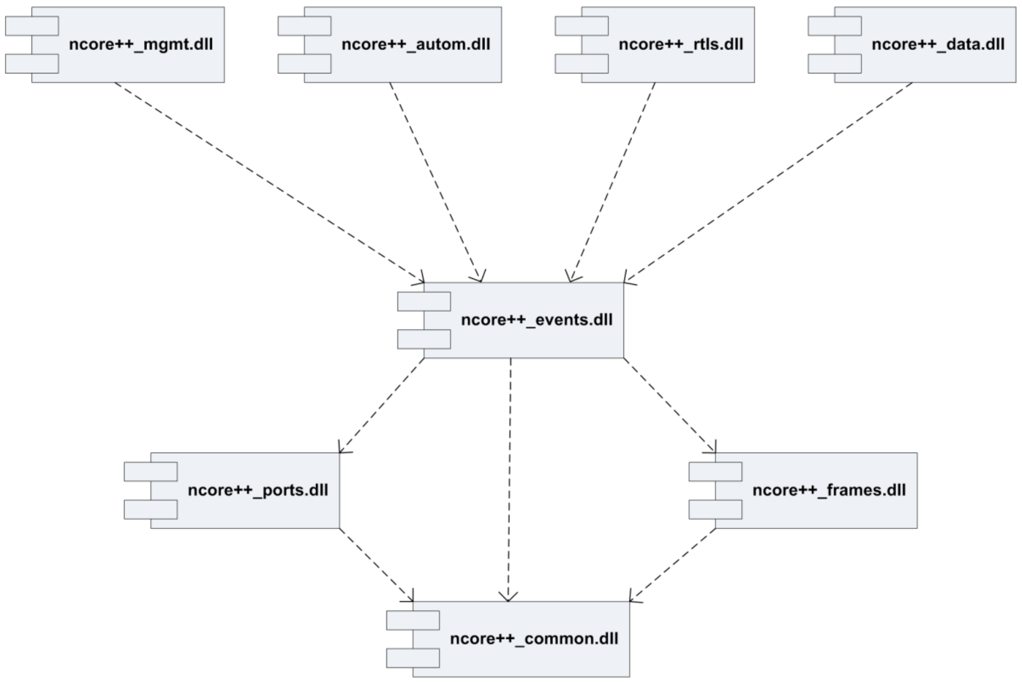


Figure 4. n-Core++ libraries.

2.2.2 n-Core.C API (ncore.c)

n-Core.C is a wrapper of n-Core++ offering its features through C functions, and is formed by five dynamic link libraries (i.e., ".*dll*" files in Windows and ".*so*" files in Linux) implementing the nine functional blocks:

- **ncore_common.dll (libncore_common.so)**
  - common, ports, frames, events, kernel
- **ncore_mgmt.dll (libncore_mgmt.so)**
  - mgmt
- **ncore_autom.dll (libncore_autom.so)**
  - autom
- **ncore_rtls.dll (libncore_rtls.so)**
  - rtls
- **ncore_data.dll (libncore_data.so)**
  - data

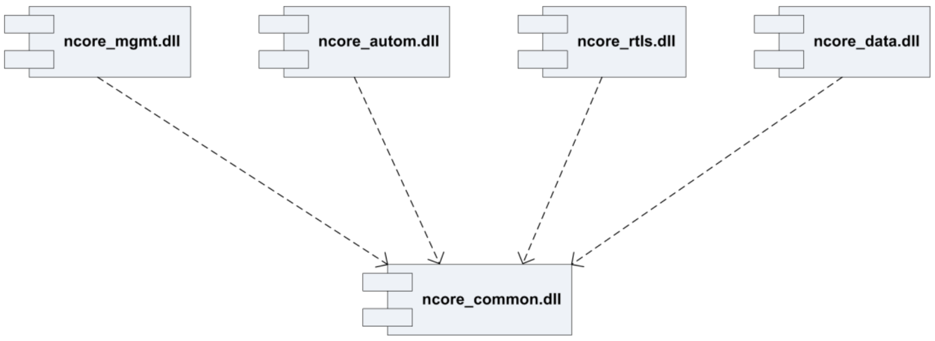


Figure 5. n-Core.C libraries.

2.2.3 n-Core.NET API (ncore.net)

n-Core.NET is a wrapper of n-Core.C offering its features through .NET classes, and is formed by five dynamic link libraries (i.e., ".*dll*" files in Windows) implementing the nine functional blocks:

- **ncorenet_common.dll**
  - common, ports, frames, events, kernel
- **ncorenet_mgmt.dll**
  - mgmt
- **ncorenet_autom.dll**
  - autom
- **ncorenet_rtls.dll**
  - rtls
- **ncorenet_data.dll**
  - data

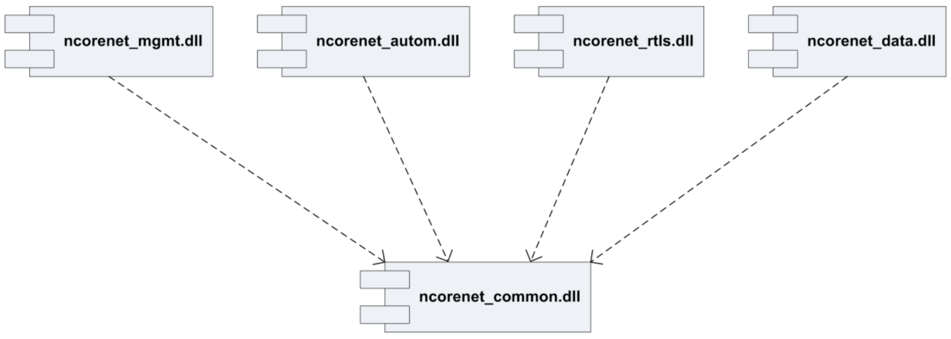
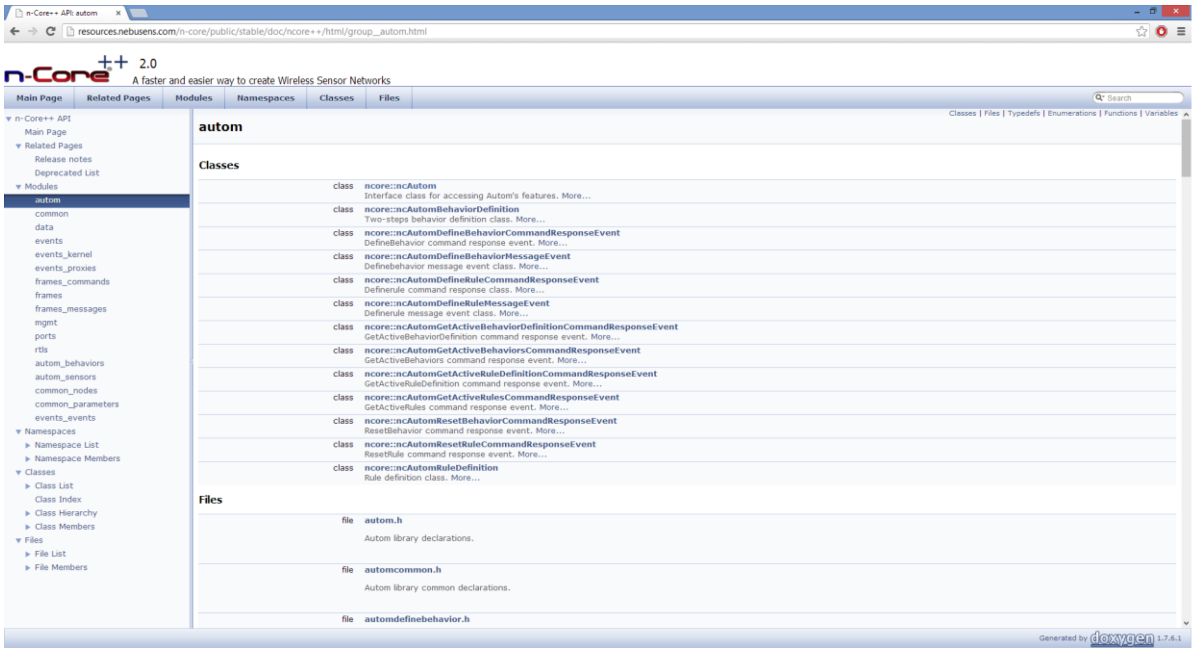


Figure 6. n-Core.NET libraries.

2.3 Documentation of n-Core® API

You can find the detailed documentation of the three n-Core® APIs in your n-Core® distribution in the next folders:

- **n-Core++**
  - *"/platform/ncore++/doc/public/doxygen/html/index.html"*
- **n-Core.C**
  - *"/platform/ncore.c/doc/public/doxygen/html/index.html"*
- **n-Core.NET**
  - *"/platform/ncore.net/doc/public/doxygen/html/index.html"*



autom

Classes

class	ncore::ncAutom Interface class for accessing Autom's features. More...
class	ncore::ncAutomBehaviorDefinition Two-steps behavior definition class. More...
class	ncore::ncAutomDefineBehaviorCommandResponseEvent DefineBehavior command response event. More...
class	ncore::ncAutomDefineBehaviorMessageEvent Definebehavior message event class. More...
class	ncore::ncAutomDefineRuleCommandResponseEvent Definerule command response class. More...
class	ncore::ncAutomDefineRuleMessageEvent Definerule message event class. More...
class	ncore::ncAutomGetActiveBehaviorDefinitionCommandResponseEvent GetActiveBehaviorDefinition command response event. More...
class	ncore::ncAutomGetActiveBehaviorsCommandResponseEvent GetActiveBehaviors command response event. More...
class	ncore::ncAutomGetActiveRuleDefinitionCommandResponseEvent GetActiveRuleDefinition command response event. More...
class	ncore::ncAutomGetActiveRulesCommandResponseEvent GetActiveRules command response event. More...
class	ncore::ncAutomResetBehaviorCommandResponseEvent ResetBehavior command response event. More...
class	ncore::ncAutomResetRuleCommandResponseEvent ResetRule command response event. More...
class	ncore::ncAutomRuleDefinition Rule definition class. More...

Files

file	autom.h Autom library declarations.
file	automcommon.h Autom library common declarations.
file	automdefinebehavior.h

Figure 7. n-Core® API online documentation.

Note: *If you do not have these files, you can download the latest n-Core® release from the private downloads area at* [*www.nebusens.com*](www.nebusens.com)

Note: *Please contact your n-Core® distributor to request your client user name and password to access the private downloads area.*

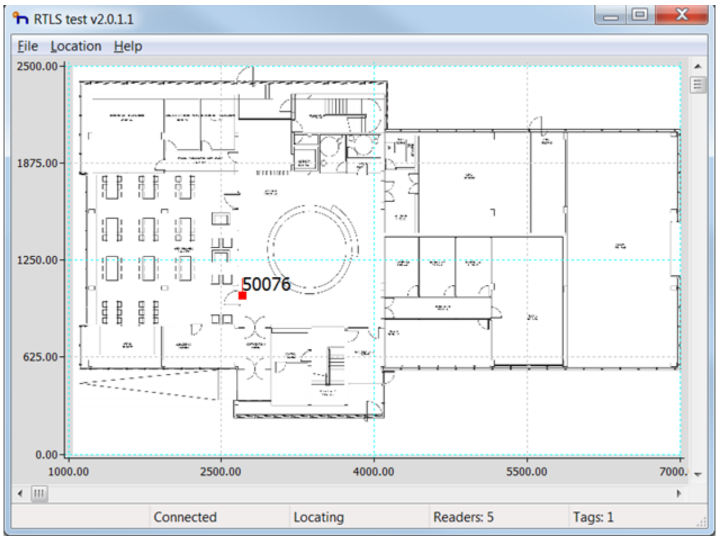
In addition, you can access the full documentation online through the next URLs:

- **n-Core++**
  - http://resources.nebusens.com/n-core/public/stable/doc/ncore++/html/
- **n-Core.C**
  - http://resources.nebusens.com/n-core/public/stable/doc/ncore.c/html/
- **n-Core.NET**
  - http://resources.nebusens.com/n-core/public/stable/doc/ncore.net/html/

2.4 Sample applications using n-Core® API

You can find some sample applications based on the three different n-Core® APIs in your n-Core® distribution in the next folders:

- **n-Core++**
  - *"/samples/ncore++"*
- **n-Core.C**
  - *"/samples/ncore.c"*
- **n-Core.NET**
  - *"/samples/ncore.net"*



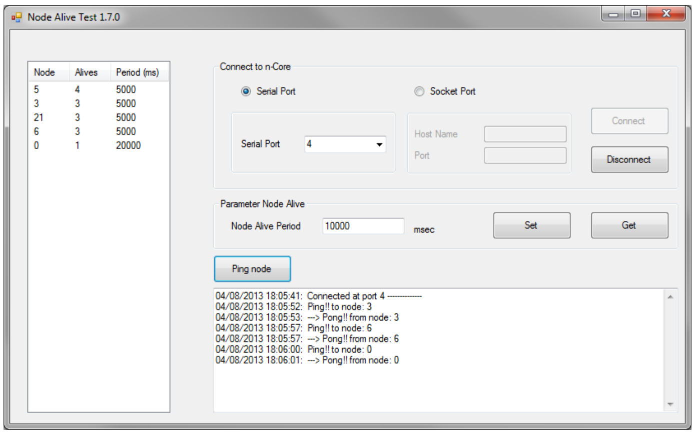
Connected | Locating | Readers: 5 | Tags: 1

Figure 8. RtlsTest sample based on the n-Core++ API.

Note: *If you do not have these files, you can download the latest n-Core® release from the private downloads area at www.nebusens.com*

Note: *Please contact your n-Core® distributor to request your client user name and password to access the private downloads area.*

These sample applications shows how to use the locating and sensing features of n-Core® in Windows and Linux systems.



Node	Alives	Period (ms)
5	4	5000
3	3	5000
21	3	5000
6	3	5000
0	1	20000

[✓ Serial Port] Socket Port

Serial Port: 4

Host Name:

Port:

Node Alive Period: 10000 msec

```
04/08/2013 18:05:41:  Connected at port 4 ------------
04/08/2013 18:05:52:  Ping!! to node: 3
04/08/2013 18:05:53:  ---> Pong!! from node: 3
04/08/2013 18:05:57:  Ping!! to node: 6
04/08/2013 18:05:57:  ---> Pong!! from node: 6
04/08/2013 18:06:00:  Ping!! to node: 0
04/08/2013 18:06:01:  ---> Pong!! from node: 0
```

Figure 9. NodeAliveTest sample based on the n-Core.NET API.

2.5 n-Core® API Initialization and n-Core® License File

It is necessary to initialize the n-Core® API before you use it in your Windows or Linux applications so that they can work properly.

Similarly, it is recommended to unitialize the n-Core® API after using it in your applications in order to free resources properly.

During the initialization, the Core® API searches and checks the n-Core® License File in the current working directory or in a path and filename specified in the corresponding initialization function.

IMPORTANT NOTE: Every n-Core® user has a unique n-Core® License File associated to his/her client ID and n-Core® Sirius devices. Therefore, it is not allowed to use devices with a different client ID.

In this sense, the three different Core® API versions (n-Core++, n-Core.C and n-Core.NET) offer appropriate functions for initializing and unitializing the libraries.

- **n-Core++**
  - *ncInitialize() and ncUnitialize() functions in "/events/events.h"*
  - *ncore++_events.dll / libncore++_events.so*
- **n-Core.C**
  - *ncCommonInitializeLibraries(), ncCommonInitializeLibrariesWithLicenseFile() and ncCommonUninitializeLibraries() in "/common/nccommon.h"*
  - *ncore_common.dll / libncore_common.so*
- **n-Core.NET**
  - *ncCommonNetInitializeLibraries(), ncCommonNetInitializeLibrariesWithLicenseFile() and ncCommonNetUninitializeLibraries() in "/nCoreCommonNet/IncludenCoreCommonNet.cs"*
  - *ncorenet_common.dll*

Furthermore, if you have a Complete n-Core® License including locating features, it is necessary to initialize RTLS libraries (see RTLS functional block) after initializing the n-Core® API and before using n-Core® features in your Windows or Linux applications so that they can work properly.

Similarly, it is recommended to unitialize the n-Core® RTLS libraries after using them in your applications in order to free resources properly.

In this regard, the three different Core® API versions (n-Core++, n-Core.C and n-Core.NET) offer appropriate functions for initializing and unitializing the RTLS libraries.

- **n-Core++**
  - *ncRtlsInitialize() and ncRtlsUnitialize() functions in "/rtls/rtls.h"*
  - *ncore++_rtls.dll / libncore++_rtls.so*
- **n-Core.C**
  - *ncRtlsInitializeLibraries() and ncRtlsUninitializeLibraries() in "/rtls/ncrtls.h"*
  - *ncore_rtls.dll / libncore_rtls.so*
- **n-Core.NET**
  - *ncRtlsNetInitializeLibraries() and ncRtlsNetUninitializeLibraries() in "/nCoreRtlsNet/IncludenCoreRtlsNet.cs"*
  - *ncorenet_rtls.dll*

2.6 Description of the different Functional Blocks

In the next sections each of the nine functional blocks of the Core® API are depicted: *Common*, *Ports*, *Frames*, *Events*, *Kernel*, *Management*, *Automation*, *RTLS* and *Data*.

2.6.1 Common Functional Block

The ***Common* functional block** includes common features to all the rest of blocks, such as basic types, classes representing the nodes in the network, classes for representing parameters that can be modified in the nodes, or logging functions.

- **n-Core++**
  - *"/common/common.h"*
  - *ncore++_common.dll / libncore++_common.so*
- **n-Core.C**
  - *"/common/nccommon.h"*
  - *ncore_common.dll / libncore_common.so*
- **n-Core.NET**
  - *"/nCoreCommonNet"*
  - *ncorenet_common.dll*

2.6.1.1 Logging features

The n-Core® API provides logging classes and functions so that the own API and the user applications can log general-purpose and debug messages, warnings, and errors to the console or to a specified file (simultaneously or not).

- **n-Core++**
  - *"/common/log.h"*
- **n-Core.C**
  - *"/common/nclog.h"*
- **n-Core.NET**
  - *"/nCoreCommonNet/ncNetLog.cs"*

2.6.1.2 Basic types

The n-Core® API provides some basic types definitions and alias widely used by the different functional blocks as Boolean values, integer values, floating-point values, vectors, matrices, spatial points and data chunks.

- **n-Core++**
  - *"/common/types.h"*
  - *"/common/chunk.h"*
  - *"/common/intpoint.h"*
  - *"/common/point.h"*
- **n-Core.C**
  - *"/common/nccommon.h"*
  - *"/autom/ncautom.h"*
  - *"/data/ncdata.h"*
  - *"/mgmt/ncmgmt.h"*
  - *"/rtls/ncrtls.h"*

- **n-Core.NET**
  - *"/nCoreCommonNet/IncludenCoreCommonNet.cs"*

- *"/nCoreAutomNet/IncludenCoreAutomNet.cs"*
- *"/nCoreDataNet/IncludenCoreDataNet.cs"*
- *"/nCoreMgmtNet/IncludenCoreMgmtNet.cs"*
- *"/nCoreRtlsNet/IncludenCoreRtlsNet.cs"*

2.6.1.3 Parameter types

Some of the functional blocks of the n-Core® API (*Kernel*, *Management*, *RTLS* and *Data*) offer configuration parameters that can be consulted remotely for each node and, in some cases, modify their value.

In this sense, the n-Core® API provides some classes/types that can be used in these *GetParam* and *SetParam* command requests and responses in those functional blocks. All these classes derive from a basic class/type, the *Parameter* class/type.

Note: *See GetParam and SetParam command requests/responses in Kernel, Management, RTLS and Data functional blocks.*

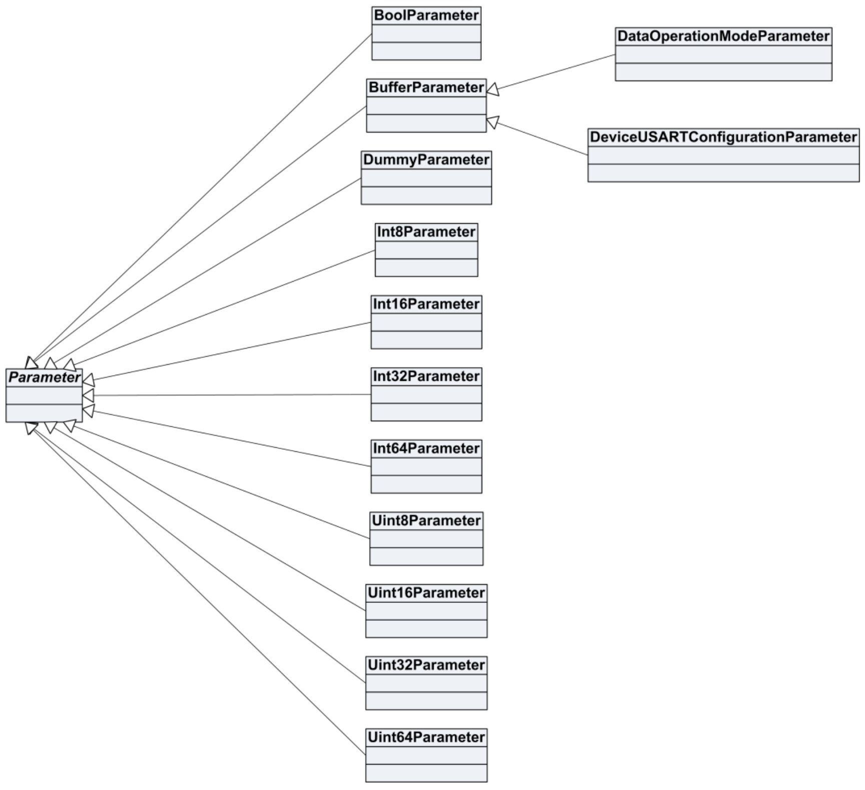


Figure 10. Parameter classes in the n-Core® API.

- **n-Core++**
  - *ncParameter and derived classes in "/common/parameters/parameters.h"*
  - *ncDeviceUSARTConfigurationParameter class in "/events/kernel/parameters/deviceusartconfigurationparameter.h" (belonging to Kernel functional block in ncore++_events.dll / libncore++_events.so)*
  - *ncDataOperationModeParameter class in "/data/parameters/dataoperationmodeparameter.h" (belonging to Data functional block in ncore++_data.dll / libncore++_data.so)*
- **n-Core.C**
  - *ncCommonParameter_t in "/common/nccommon.h" (belonging to Common functional block in ncore_common.dll / libncore_common.so)*

  - *ncDeviceUSARTConfigurationParameter_t in "/common/nccommon.h" (belonging to Common functional block in ncore_common.dll / libncore_common.so)*
  - *ncDataOperationModeParameter_t in "/data/ncdata.h" (belonging to Data functional block in ncore_data.dll / libncore_data.so)*
- **n-Core.NET**
  - *ncCommonNetParameter and derived classes in "/nCoreCommonNet/Parameters" included ncCommonNetDeviceUSARTConfigurationParameter (belonging to Common functional block in ncorenet_common.dll)*
  - *ncDataNetOperationModeParameter in "/nCoreDataNet/Parameters" (belonging to Data functional block in ncorenet_data.dll)*

2.6.1.4 Node types

Most of the functional blocks of the n-Core® API offer features related to nodes forming a wireless sensor network, that is, the n-Core® Sirius devices.

In this sense, the *Common* functional block of the n-Core® API provides some classes/types that are used by the rest of the functional blocks in their command request/responses, messages and events.

All these classes derive from a basic class/type, the *Node* class/type.

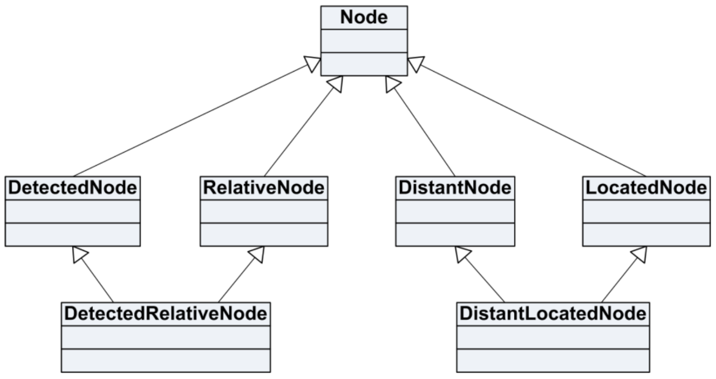


Figure 11. Node classes in the n-Core® API.

These node classes/types make use of other classes/types representing spatial points (See *Basic types* in this functional block) or other classes/types such as the *DetectionValue* and derived classes, amongst others.

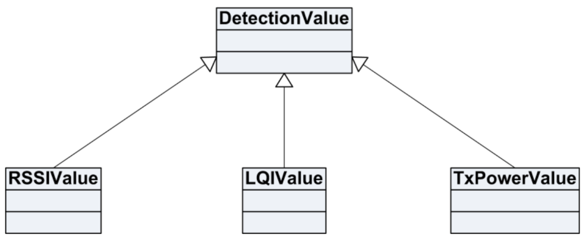


Figure 12. Detection value classes in the n-Core® API.

- **n-Core++**
  - *ncNode and derived classes in "/common/nodes/nodes.h"*
  - *ncDetectionValue and derived classes in "/common/nodes/nodes.h"*

2.6.2 Ports Functional Block

The ***Ports* functional block** includes classes/types intended for representing communication ports through which the n-Core® API based applications can communicate with a n-Core® Sirius device that acts as gateway to a n-Core® network, usually a coordinator or a collecting node.

These communication ports can be serial com ports (i.e., COM ports) or socket ports (i.e., TCP/IP connections).

- **n-Core++**
  - *"/ports/ports.h"*
  - *ncore++_ports.dll / libncore++_ports.so*
- **n-Core.C**
  - *ncCommonSerialComPort_t and ncCommonSocketClientPort_t in "/common/nccommon.h"*
  - *ncore_common.dll / libncore_common.so*
- **n-Core.NET**
  - *ncCommonNetSerialComPort and ncCommonNetSocketClientPort classes derived from ncCommonNetSocketPort class in "/nCoreCommonNet/CommPorts"*
  - *ncorenet_common.dll*

All these classes representing communication ports derive from a basic class/type, the *CommPort* class/type.

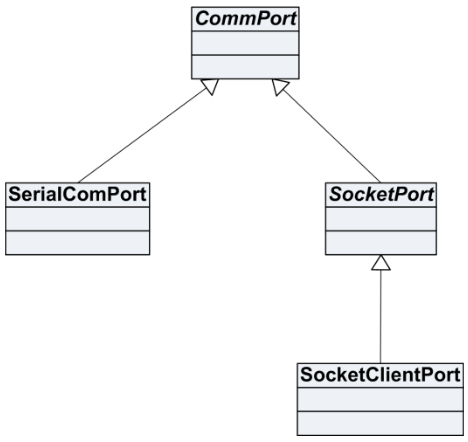


Figure 13. Communication port classes in the *Ports* functional block.

These *CommPort* derived classes/types are used along with *Protocol* derived classes/types and both the communication port and the protocol are automatically managed by the n-Core® API by means of *Proxy* classes/types.

Note: *See Protocol classes/types in Frames functional block and Proxy classes/types in Events functional block.*

There are some parameters common to all *CommPort classes:*

- *timeout*: timeout elapsed before the API considers that a connection or a reconnection with a communication port has been not achieved.
- *autoReconnect*: if the API should reconnect automatically to the communication port if there is disconnected on the other side.
- *deferredConnection*: if the API allows the connection to the port in a deferred way, that is, not blocking the application code when trying to connect to the port.

Connections and reconnections are notified to user application code by means of ProxyEvents (See Events functional block).

- **n-Core++**
  - *ncCommPort class in "/ports/commport.h"*
- **n-Core.C**
  - *ncCommonSerialComPort_t and ncCommonSocketClientPort_t in "/common/nccommon.h"*
  - *ncore_common.dll / libncore_common.so*
- **n-Core.NET**
  - *ncCommonNetSerialComPort and ncCommonNetSocketClientPort classes derived from ncCommonNetSocketPort class in "/nCoreCommonNet/CommPorts"*
  - *ncorenet_common.dll*

2.6.2.1 Serial com ports

All n-Core® Sirius devices include an USART chipset that allows transmissions between them and the computer or other device connected to it through a COM port using a USB (virtual COM port in this case, for Sirius A/B/D devices directly, or for Sirius Quantum/RadIOn devices using a Sirius IOn-D device) or a RS-232 cable (only present in Sirius A devices).

The n-Core® API provides the *SerialComPort* class/type to represent and manage these serial com ports. The parameters specific to *SerialComPort* class/type are:

- *name*: the serial com port name in the system. For example, "com3" or "\\\\.\\com64" in Windows or "/dev/ttyS0" or "/dev/ttyUSB0" in Linux systems.
- *baudrate*: baud rate used in the transmissions (by default is "38400").
- *bitsPerChar*: bits per character used in the transmission (by default is "8").
- *parity*: parity bits used in the transmissions (by default is "NONE").
- *stopbits*: stop bits used in the transmissions (by default is "1").
- *xonxoff*: software flow control (XON/XOFF) (by default is "false").
- *rtscts*: hardware flow control (RTS/CTS) (by default is "false").

Note: *make sure you have installed the n-Core® Sirius devices' driver in your Windows or Linux operating system. This driver is in the "\tools\cp210x" folder in your n-Core® distribution. Linux distributions including 2.6 series kernel (≥ 2.6.12) usually provide CP210X driver as part of the default installation. You can also download the "CP210x USB to UART Bridge VCP Drivers" from the website of Silicon Laboratories at www.silabs.com*

- **n-Core++**
  - *ncSerialComPort class in "/ports/serialcomport.h"*
- **n-Core.C**
  - *ncCommonSerialComPort_t in "/common/nccommon.h"*
  - *ncore_common.dll / libncore_common.so*
- **n-Core.NET**
  - *ncCommonNetSerialComPort class in "/nCoreCommonNet/CommPorts"*
  - *ncorenet_common.dll*

2.6.2.2 Socket ports

In large or remote n-Core® sensing and/or locating applications, users can have a set of collecting nodes that forward sensing and detection information to the n-Core® API running linked to the user application in a computer. These collecting nodes can be connected directly through USB or RS-232 ports or by means of

TCP/IP sockets. In the last case users can utilize n-Core® Sirius A devices connected to UART-TCP converters as collecting nodes. There are versions for Ethernet, Wi-Fi or even GPRS for these UART-TCP converters.

In order to manage these kinds of remote collecting nodes, the n-Core® API provides the *SocketClientPort* class/type, that allows specifying a set of remote IP address and TCP port to which the API can communicate.

- **n-Core++**
  - *ncSocketClientPort class in "/ports/socketclientport.h"*
- **n-Core.C**
  - *ncCommonSocketClientPort_t in "/common/nccommon.h"*
  - *ncore_common.dll / libncore_common.so*
- **n-Core.NET**
  - *ncCommonNetSocketClientPort class derived from ncCommonNetSocketPort class in "/nCoreCommonNet/CommPorts"*
  - *ncorenet_common.dll*

2.6.3 Frames Functional Block

The ***Frames* functional block** includes classes/types intended for representing protocols to be used by the n-Core® API based applications to communicate with a n-Core® Sirius device that acts as gateway to a n-Core® network, usually a coordinator or a collecting node.

Currently there is only provided the n-Core® protocol to communicate with n-Core® Sirius devices.

- **n-Core++**
  - *"/frames/frames.h"*
  - *ncore++_frames.dll / libncore++_frames.so*
- **n-Core.C**
  - *"/common/nccommon.h"*
  - *ncore_common.dll / libncore_common.so*
- **n-Core.NET**
  - *"/nCoreCommonNet"*

2.6.3.1 Protocols and n-Core® Protocol

All classes representing protocols derive from a basic class/type, the *Protocol* class/type.

The current protocol used for communicating with n-Core® Sirius devices is the n-Core® Protocol version 2.0, represented by the *CoreProtocol* class/type.

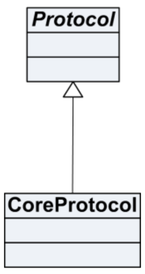


Figure 14. Protocol classes in the *Frames* functional block.

These *Protocol* derived classes/types are used along with *CommPort* derived classes/types and both the communication port and the protocol are automatically managed by the n-Core® API by means of *Proxy* classes/types.

Note: *See CommPort classes/types in Ports functional block and Proxy classes/types in Events functional block.*

- **n-Core++**
  - *ncProtocol class in "/frames/protocol.h"*
  - *ncCoreProtocol class in "/frames/ncoreprotocol.h"*
- **n-Core.C**
  - *ncCommonProtocolId_t and ncCommonProtocolVersion_t in "/common/nccommon.h"*
- **n-Core.NET**
  - *ncCommonNetProtocol class in "/nCoreCommonNet/CommPorts/ncCommonNetProtocol.cs"*
  - *ncCommonCoreNetProtocol class in "/nCoreCommonNet/CommPorts/ncCommonCoreNetProtocol.cs"*

2.6.4 Events Functional Block

The ***Events*** **functional block is the core functional block of the n-Core® API.** This block provides *Proxy* classes/types to manage a certain communication port connected to a n-Core® Sirius device that acts as gateway to a n-Core® network, usually a coordinator or a collecting node, and using a certain protocol.

Developers can send command request and messages through a ***Proxy*** class/type and receive command responses, messages, as well as location information and other proxy events, by means of ***Event*** *classes/types* provided from ***EventHandler*** objects that are registered in *Proxy* objects according to the user requirements.

Note: *It is highly recommended to explore the different samples provided in the n-Core® distribution in order to know how use proxies, event handlers/dispatchers and events when developing n-Core® sensing and locating applications.*

- **n-Core++**
  - *"/events/events.h"*
  - *ncore++_events.dll / libncore++_events.so*
- **n-Core.C**
  - *There are different events for each module*
- **n-Core.NET**
  - *There are different events for each module but all derived from ncCommonNetEvent class in "/nCoreCommonNet/Events/ncCommonNetEvent.cs"*

2.6.4.1 Proxies

Protocol derived classes/types are used along with *CommPort* derived classes/types and, thus, both the communication port and the protocol are automatically managed by the n-Core® API by means of *Proxy* classes/types.

n-Core® based applications can manage a *Proxy or a* set of *Proxies* according to their needs. For example, applications can open a set of remote TCP/IP connections using a set *Proxies*, *CoreProtocol* and *SocketClientPorts* objects in order to run a certain *Locater* (*See Locaters in the RTLS functional block*) and receive *LocationEvents* using a certain *EventHandler* or *EventDispatcher*.

Note: *See CommPort classes/types in Ports functional block and Protocol classes/types in Frames functional block.*

The basic features of *Proxy* classes/types are:

1. Opening the associated communication port (***OpenPort***).
2. Closing the associated communication port (***ClosePort***).
3. Starting to listen on the associated communication port (***Listen***).
   - *It is necessary to listen on the associated communication port in order to receive any kind of event coming from the proxy.*
4. Stopping listening on the associated communication port (***StopListen***).
5. Getting the id of the associated node (***GetAssociatedNodeId***).
6. Registering a handler to receive proxy events (***RegisterProxyEventHandler***).
7. Unregistering a handler to receive proxy events (***UnregisterProxyEventHandler***).

Note: *See Events and Event Handlers in this functional block.*

In addition to these features, *Proxy* objects are used in the functions/methods in the *Kernel*, *Management*, *Automation, RTLS* and *Data* functional blocks in order to access the n-Core® nodes using the featured provided by such blocks.

- **n-Core++**
  - *ncProxy class in "/events/proxies/proxies.h"*
- **n-Core.C**
  - *ncCommonProxy_t in "/common/nccommon.h"*
- **n-Core.NET**
  - *ncCommonNetProxy class in "/nCoreCommonNet/proxy/ncCommonNetProxy.cs"*

2.6.4.2 Events and Event Handlers

Developers can receive command responses, messages, as well as location information and other proxy events, by means of *Event classes/types* provided from *EventHandler* objects that are registered in *Proxy* objects according to the user requirements.

All events in n-Core® derive from a basic class/type, the *Event* class/type. In the rest of functional blocks (*Kernel*, *Management*, *Automation, RTLS* and *Data* functional) there are new events derived from the classes/types shown in the Figure 15.

The special *ProxyEvent* class/type is directly associated to the *Proxy* itself and provides information related to the changes in the state of the connection through it:

1. ***CONNECTED:*** *Port connected (the port was connected after a successful non-deferred open or, a successful deferred connection or a successful autoreconnect).*
2. ***DISCONNECTED:*** *Port disconnected (port is disconnected and it will not try to autoreconnect) or the user calls Proxy::ClosePort.*
3. ***AUTORECONNECTING:*** *Autoreconnect timeout (the port is detected as disconnected and is going to try an autoreconnection).*

Note: *See the RTLS functional block for more information about LocationEvents.*

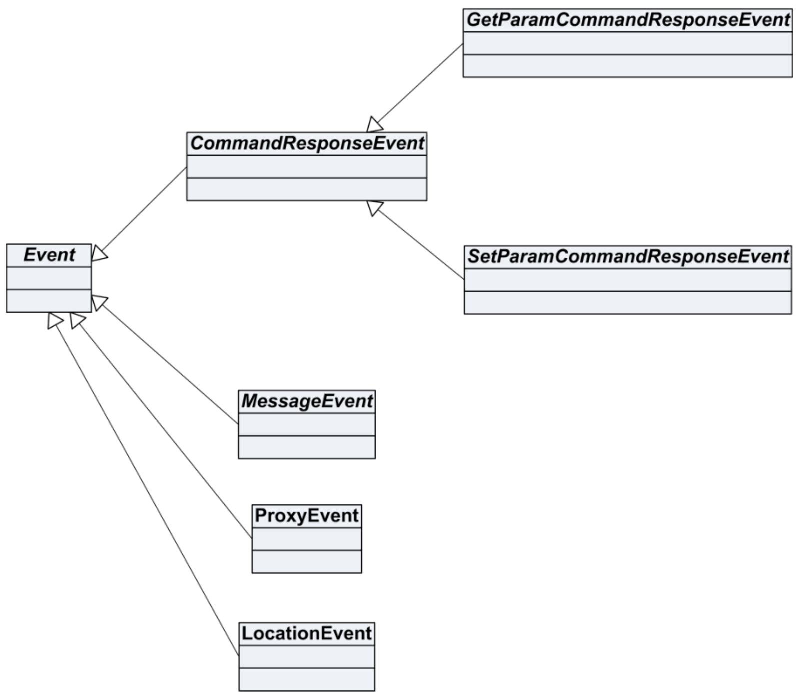


Figure 15. Event classes in the *Events* functional block.

- **n-Core++**
  - *ncEvent and derived classes in "/events/events/events.h"*
  - *ncProxyEvent class in "/events/proxies/proxyevent.h"*
  - *ncLocationEvent class in "/rtls/locater/locationevent.h" (belonging to RTLS functional block in ncore++_rtls.dll / libncore++_rtls.so)*
- **n-Core.C**
  - *ncCommonProxyEvent_t in "/common/nccommon.h"*
  - *ncRtlsLocationEvent_t in "/rtls/ncrtls.h" (belonging to RTLS functional block in ncore_rtls.dll / libncore_rtls.so)*
- **n-Core.NET**
  - *ncCommonNetEvent class in "/nCoreCommonNet/Events/ncCommonNetEvent.cs"*
  - *ncCommonNetProxyEvent class in "/nCoreCommonNet/Events/ncCommonNetProxyEvent.cs"*
  - *ncRtlsLocationEvent class in "/nCoreRtlsNet/Events/ncRtlsNetLocationEvent.cs"*

In order to the user code can manage these *Events*, the n-Core® API provides *EventHandler* classes/types that can be registered in *Proxy* objects according to the user requirements using the different features offered by the different functional blocks of the API.

Developers can write their callback code to receive these events, both writing their own classes derived from the abstract *EventHandler* class, implementing its *ProcessEvent* method as a callback function (e.g., in the case of n-Core++ API), or providing a callback function (e.g., in the case of n-Core.C API).

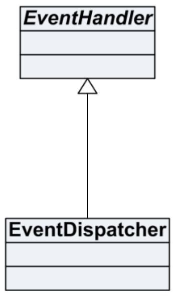


Figure 16. Event handler classes in the *Events* functional block.

Developers can also make use of the *EventDispatcher* class in any place that *EventHandler* can be used (as *EventDispatcher* derives from *EventHandler*). An *EventDispatcher* object accumulates received *Events* so that developers can "pop" pending *Events* when they decide in their code (e.g., using a periodic timer). The basic features of *EventDispatcher* classes/types (in addition to *EventHandler*) are:

1. Guessing if are pending events (and how many) (***ArePendingEvents***).
2. Obtaining (cloning) the next pending event (not removing it) (***GetNextEvent***).
3. Popping the next pending event from the queue (and deleting it) (***PopNextEvent***).
4. Clearing all pending events (***ClearEvents***).

- **n-Core++**
  - *ncEventHandler class in "/events/events/eventhandler.h"*
    - *In n-Core++, developer must write a class derived from abstract ncEventHandler class, implementing his/her own ProcessEvent() method (or use directly ncEventDispatcher class).*
  - *ncEventDispatcher class in "/events/events/eventdispatcher.h"*
- **n-Core.C**
  - *There are different EventHandler per type of event*
  - *There are different EventDispatcher for each module (see also nc**Common**CreateEvtDispatcher, nc**Autom**CreateEvtDispatcher, etc.)*
- **n-Core.NET**
  - *ncCommonNetEventHandler class in "/nCoreCommonNet/Events/ncCommonNetEventHandler.cs"*
    - *In n-Core.Net, developer must write a class derived from abstract ncCommonNetEventHandler class, implementing his/her own ProcessEvent() method (or use directly nc**XXX**NetEventDispatcher classes).*
  - *There are different event dispatchers for each module:*
    - *ncCommonNetEventDispatcher class in "/nCoreCommonNet/Events/ncCommonNetEventDispatcher.cs"*
    - *ncAutomNetEventDispatcher class in "/nCoreAutomNet/Events/ncAutomNetEventDispatcher.cs"*
    - *ncMgmtNetEventDispatcher class in "/nCoreMgmtNet/Events/ncMgmtNetEventDispatcher.cs"*
    - *ncDataNetEventDispatcher class in "/nCoreDataNet/Events/ncDataNetEventDispatcher.cs"*
    - *ncRtlsNetEventDispatcher class in "/nCoreRtlsNet/Events/ncRtlsNetEventDispatcher.cs"*

2.6.5 Kernel Functional Block

The ***Kernel* functional block** provides some basic functionalities of the platform that cannot be included inside other functional block because of its nuclear behavior.

- **n-Core++**
  - *"/events/kernel/kernel.h"*
  - *ncore++_events.dll / libncore++_events.so*
- **n-Core.C**
  - *"/common/nccommon.h"*
  - *ncore_common.dll / libncore_common.so*
- **n-Core.NET**
  - *"/nCoreCommonNet"*

2.6.5.1 Configuration parameters of the Kernel functional block

The configuration parameters of the *Kernel* functional block can be consulted remotely for each node and, in some cases, modify their value. These parameters are the next ones:

1. ***Coordinate* X (*Coord X*):** coordinate X of the node.
2. ***Coordinate* Y (*Coord Y*):** coordinate Y of the node.
3. ***Coordinate* Z (*Coord Z*):** coordinate Z of the node.

   Note: *these three parameters (coordinate X, coordinate Y and coordinate Z) do not have a predefined longitudinal metric unit. The user is responsible for choosing the best and the most convenient longitudinal metric unit, bearing in mind he/she has to be consistent with his/her choice. The user must always use the same units for all the nodes and all the dimensions. The user has to consider those units to configure the different RTLS algorithms, because these algorithms will show their results in the same units the user has chosen for the nodes. For example, if the user establishes the coordinates of all the nodes in centimeters, then the coordinates of the located nodes after the location algorithms will be shown in centimeters.*

   IMPORTANT NOTE: these three parameters (coordinate X, coordinate Y and coordinate Z) are here in Kernel functional block instead of RTLS block since n-Core® 2.0.

4. **Device state (*Device State*):** Node working mode: *setup* mode or *work* mode.
   - ***work* mode** is the usual operation mode for a Sirius device.
   - In ***setup* mode**, the device will attend all the commands sent to it, including configuration commands, and will modify its internal configuration basing on the received requests. However, the device will not perform any *behavior* or *rule* (see Autom functional block) and will not send any kind of RTLS message. The only messages sent by the device will be *NodeIdNotify* and *NodeAlive* (see messages in this section) and Data messages (see messages in Data functional block).

   IMPORTANT NOTE: this parameter is here in Kernel functional block instead of Mgmt block since n-Core® 2.0.

5. **Device USART configuration (*Device USART Configuration*):** pack of configuration parameters of the USART chipset responsible for transmissions between the Sirius device and the computer or other device connected to it through a COM port using a USB (virtual COM port in this case) or a RS-232 cable (only present in Sirius A devices).
   - channel: USART channel in the device **("1" in Sirius A/B/D devices, and "0" in Sirius Quantum/RadIOn devices**).

- synchronous mode: if the device will use synchronous or asynchronous transmission mode.
- baudrate: baud rate used in the transmissions.
- bitsPerChar: bits per character used in the transmission.
- parity: parity bits used in the transmissions.
- stopbits: stop bits used in the transmissions.
- xonxoff: software flow control.
- rtscts: hardware flow control.

IMPORTANT NOTE: not all the combinations of parameters are allowed. Please, consult the n-Core® API documentation.

- **n-Core++**
  - *ncKernelParamId type in "/events/kernel/kernelparamstypes.h"*
- **n-Core.C**
  - *ncCommonParamId_t in "/common/nccommon.h"*
- **n-Core.NET**
  - *ncCommonNetParamId in "/nCoreCommonNet/IncludenCoreCommonNet.cs"*

In Table 1, the *Kernel* parameters and their default values are shown.

Table 1. Default values for the configuration parameters of the *Kernel* functional block.

Parameter	Default value	Read-only
Coord X	special "undefined" value	-
Coord Y	special "undefined" value	-
Coord Z	special "undefined" value	-
Device State	*Setup* mode	-
Device USART Configuration	• channel = 1 (Sirius A/B/D) or 0 (Sirius Quantum/RadIOn) • synchronousMode = false • baudrate = 38400 • bitsPerChar = 8 • parity = none • stopbits = 1 • xonxoff = false • rtscts = false	-

2.6.5.2 Kernel command request/response frames

The command request and command response frames that exist in this block are the next ones:

1. **Obtaining a Kernel configuration parameter of a node (*GetParam*)**: it allows obtaining the value of a configuration parameter of a node connected to the ZigBee™ network.
   - **n-Core++:**
     - *ncKernel::GetParam() method in "/events/kernel/kernel.h"*
     - *ncKernelGetParamCommandResponseEvent class*
   - **n-Core.C:**
     - *ncCommonGetParam function in "/common/nccommon.h"*
     - *ncCommonGetParamCommandResponseEvent_t struct*
   - **n-Core.NET:**
     - *ncCommonNetGetParam in "/nCoreCommonNet/IncludenCoreCommonNet.cs"*
     - *ncCommonNetGetParamCommandResponseEvent class*
2. **Setting or modifying a Kernel configuration parameter of a node (*SetParam*)**: this command allows the user to modify the value of a configuration parameter of a node connected to the ZigBee™ network.

- **n-Core++:**
  - *ncKernel::SetParam() method in "/events/kernel/kernel.h"*
  - *ncKernelSetParamCommandResponseEvent class*
- **n-Core.C:**
  - *ncCommonSetParam function in "/common/nccommon.h"*
  - *ncCommonSetParamCommandResponseEvent_t struct*
- **n-Core.NET:**
  - *ncCommonNetSetParam in "/nCoreCommonNet/IncludenCoreCommonNet.cs"*
  - *ncCommonNetSetParamCommandResponseEvent class*

2.6.5.3 Kernel message frames

The message frames that exist in this block are:

1. **Node identification notifying message (*NodeIdNotify*)**: this message allows the API to be notified automatically or on demand about the network identification (*Node Id*) of the device connected through the serial port to the computer where the n-Core® API is running.
   - **n-Core++:**
     - *ncKernel::NodeIdNotifyRequest() in "/events/kernel/kernel.h"*
     - *ncKernel::RegisterNodeIdNotifyEventHandler() in "/events/kernel/kernel.h"*
     - *ncKernel::UnregisterNodeIdNotifyEventHandler() in "/events/kernel/kernel.h"*
     - *ncKernelNodeNotifyMessageEvent class*
   - **n-Core.C:**
     - *ncCommonNodeIdNotifyRequest in "/common/nccommon.h"*
     - *ncCommonRegisterNodeIdNotifyEventHandler in "/common/nccommon.h"*
     - *ncCommonUnregisterNodeIdNotifyEventHandler in "/common/nccommon.h"*
     - *ncCommonRegisterNodeIdNotifyEventDispatcher in "/common/nccommon.h"*
     - *ncCommonUnregisterNodeIdNotifyEventDispatcher in "/common/nccommon.h"*
     - *ncCommonNodeIdNotifyMessageEvent_t struct*
   - **n-Core.NET:**
     - *ncNetNodeIdNotifyRequest in "/nCoreCommonNet/ncNetKernel.cs"*
     - *ncNetRegisterNodeIdNotifyEventHandler in "/nCoreCommonNet/ncNetKernel.cs"*
     - *ncNetUnregisterNodeIdNotifyEventHandler in "/nCoreCommonNet/ncNetKernel.cs"*
     - *ncNetRegisterNodeIdNotifyEventDispatcher in "/nCoreCommonNet/ncNetKernel.cs"*
     - *ncNetUnregisterNodeIdNotifyEventDispatcher in "/nCoreCommonNet/ncNetKernel.cs"*
     - *ncCommonNetNodeIdNotifyMessageEvent class*

2. **Node Alive Message (*NodeAlive*):** this message frame is sent to a collecting node periodically by all those *router* or *tag* nodes that have defined the node alive period (see *Alive Period* in Mgmt functional block) to a value different from zero. The collecting node forwards these kinds of messages to the computer connected through its USB or RS-232 serial port connection.

   This message contains the next information from the node that transmits the Node Alive message to the collecting node:
   - Device state (see *Device State* parameter in this functional block).
   - Node id (see *Node Id* parameter in Mgmt functional block).
   - UID (see *UID* parameter in Mgmt functional block).
   - Node Alive period (see *Alive Period* parameter in Mgmt functional block).
   - Collecting node id (see *Collecting Node Id* parameter in Mgmt functional block).
   - Node position (see *Coord X*, *Coord Y* and *Coord Z* parameter in this functional block).

- **n-Core++:**
  - *ncKernel::RegisterNodeAliveEventHandler() method in "/events/kernel/kernel.h"*
  - *ncKernel::UnregisterNodeAliveEventHandler() method in "/events/kernel/kernel.h"*
  - *ncKernelNodeAliveMessageEvent class*
- **n-Core.C:**
  - *ncCommonRegisterNodeAliveMessageEventHandler in "/common/nccommon.h"*
  - *ncCommonUnregisterNodeAliveMessageEventHandler in "/common/nccommon.h"*
  - *ncCommonRegisterNodeAliveMessageEventDispatcher in "/common/nccommon.h"*
  - *ncCommonUnregisterNodeAliveMessageEventDispatcher in "/common/nccommon.h"*
  - *ncCommonNodeAliveMessageEvent_t struct*
- **n-Core.NET:**
  - *ncCommonNetRegisterNodeAliveMessageEventHandler in "/nCoreCommonNet/IncludenCoreCommonNet.cs"*
  - *ncCommonNetUnregisterNodeAliveMessageEventHandler in "/nCoreCommonNet/IncludenCoreCommonNet.cs"*
  - *ncCommonNetRegisterNodeAliveMessageEventDispatcher in "/nCoreCommonNet/IncludenCoreCommonNet.cs"*
  - *ncCommonNetUnregisterNodeAliveMessageEventDispatcher in "/nCoreCommonNet/IncludenCoreCommonNet.cs"*
  - *ncCommonNetNodeAliveMessageEvent class*

IMPORTANT NOTE: The node position message is in the Kernel functional block instead of the Mgmt block since n-Core® 2.0.

2.6.6 Management Functional Block

The ***Management* functional block** is composed by all the functionalities related to the configuration or modification of the basic network parameters. These parameters affect the node behavior regarding the ZigBee™ network.

- **n-Core++**
  - *"/mgmt/mgmt.h"*
  - *ncore++_mgmt.dll / libncore++_mgmt.so*
- **n-Core.C**
  - *"/mgmt/ncmgmt.h"*
  - *ncore_mgmt.dll / libncore_mgmt.so*
- **n-Core.NET**
  - *"/nCoreMgmtNet"*
  - *ncorenet_mgmt.dll*

2.6.6.1 Configuration parameters of the Management functional block

The configuration parameters of the *Management* functional block can be consulted remotely for each node and, in some cases, modify their value. These parameters are the next ones:

1. **Unique IDentification (*UID*)**: node unique identification. This UID is the device MAC and it is tied to the hardware. Read-only parameter.
2. **Node Network Address (*Node Id*)**: node address in the ZigBee™ network. This node identification must be unique for every node in the ZigBee™ network in order that inconsistencies do not exist in the infrastructure.

Note: see Annex **¡Error! No se encuentra el origen de la referencia.** *for the specific Node id values according to the type of device.*

3. **Transmission Power (*Tx Power*)**: Transmission power that the node uses to transmit its signal. Its value depends on the frequency band used for the transmissions.
   - **Frequency Band 2.4 GHz**: power available values vary between -17 dBm and +3 dBm.

     Note: *these values are the ones the user can select using the n-Core® software, but the 2.4 GHz n-Core® Sirius devices are hardware amplified. The device output real value will be higher than the value select by software and depends on the type of Sirius used:*
     - *Sirius A, B and D: the value selected by software will be incremented in +12 dBm.*
     - *Sirius Quantum and RadIOn: the value selected by software will be incremented in +22 dBm.*

   - **Frequency Band 900 MHz (868 MHz/915 MHz):** power available values vary between -11 dBm and +5 dBm (*normal mode*)[1].

4. **Neighbor Table Size (*Neib Table Size*)**: Maximum number of neighbors that a *node can h*ave in the Zigbee™ network. Read-only parameter.
5. **Maximum number of children (*Max Children Amount*)**: Maximum number of children that a node can have in the ZigBee™ network. Read-only parameter.
6. **Maximum number of router children *(Max Children Router Amount)***: Maximum number of routers that can be children of a node in the ZigBee™ network. Read-only parameter.
7. **Maximum network depth (*Max Network Depth*)**: Maximum number of hops in a ZigBee™ network. Read-only parameter.
8. **Collecting Node Network Address (*Collecting Node*)**: Node address where some messages are sent. For example, the *Node Alive* message in the Kernel block or the *Tags Table* messages in the RTLS functional block. It is not necessary that all the nodes of the ZigBee™ network have the same collecting node, as the API can gather data from a set of collecting nodes (see *Proxies* in Events functional block). By default, the collecting node of all the nodes is the coordinator node of the ZigBee™ network.

   These are the messages that a node can send to its collecting node according to the functional block:
   - Kernel: *Node Alive (if Node Alive period in Mgmt block is different from zero)*
   - Autom: *Define Behavior message (if enabled in behavior definition), Define Rule message (if enabled in rule definition)*
   - RTLS: *Tags Table message (if Tags Table period in RTLS block is different from zero and the device is not a Tag)*

9. **Node alive period (*Alive Period)***: Period used to send the *NodeAlive* message in the Kernel functional block to the collecting node periodically. By default, this parameter is configured to 10000 ms. It is advisable that this value should not have too low values and that it will not be lower than 2000 ms, since it can affect the performance of the network.

[1] If *boost mode* is used, power transmission value can get to +11dBm. Please, look up the restrictions and local regulations for this power transmission.

IMPORTANT NOTE: *Device State parameter is not anymore here in Mgmt functional block, as it has been moved to Kernel block since n-Core®* 2.0.

10. **Hardware type (*Hardware Type*)**: Defines the node at hardware level: *Sirius A, Sirius B/D or Sirius Quantum/RadIOn*. Read-only parameter.
11. **Application device type (*App Device Type*)**: Defines the node at application level. A node can get one of the next roles in the ZigBee™ network:
    - ***Coordinator***: Creates the ZigBee™ network. Its network address (*Node id*) must be always 0 and it is important that one and only one coordinator is present in the same ZigBee™ network in order to avoid failures sending and receiving frames. These devices act also as collecting nodes and can also act as readers in the locating infrastructure.
    - ***Router*****:** These devices are used as fixed devices in ZigBee™ network, forwarding frames to another devices and also acting as readers in the locating infrastructure.
    - ***Tag*****:** These devices are generally used as devices that need low-energy consumption. They are usually battery-supplied and they are used as mobile devices in the ZigBee™ network.
    - **Collecting node**: These devices collect messages from other nodes of ZigBee™ network and forwards to the computer/device connected to it through a COM port (see *Collecting Node* parameter in this functional block). These devices act also as routers in the network and can also act as readers in the locating infrastructure.

      Note: *see Annex* **¡Error! No se encuentra el origen de la referencia.** *for the specific Node id values according to the type of device.*

12. **Application firmware version (*Firmware Version*).** Unsigned 32-bit value representing the version of the n-Core® firmware inside the Sirius device. Read-only parameter.
13. **Sleep period of the device (*Sleep Period*).** Unsigned 32-bit value representing the sleep period (only applicable to Tags, see *App Device Type* in this functional block), of the device (in milliseconds).

    Note: *In Tags, in order that the device works properly, the Sleep Period must be lower than some other periods in the device: Alive Period in Mgmt, Broadcast Period in Mgmt, as well as the periods established in behaviors and rules in the Tag (see Behaviors and Rules in the Autom functional block).*

    Note: *See Annex* B *where the facts about the Sleep Period are detailed.*

14. **Awake period of the device (*Awake Period*).** Unsigned 32-bit value representing the awake period (only applicable to Tags, see *App Device Type* in this functional block), of the device (in milliseconds). The Awake Period is the time during which a router will stay awake before going to sleep again. This parameter will be enabled only when the sleep period parameter of the device is different from zero.
15. **Broadcast period (*Broadcast Period*):** it is the period used by the *Tags*, *Routers and Mobile Routers* to send the RTLS broadcast frames to the rest of devices connected to the network (see Broadcast messages in RTLS functional block). This value is also considered for RSSI port behaviors and rules (see *Define Behavior* and *Define Rule* messages in Autom functional block). **It is strongly recommended not to establish values lower than 250 ms.**

    IMPORTANT NOTE: *this parameter is in the Mgmt functional block instead of the RTLS block since n-Core®* 2.0.

16. **Minimum RSSI value (*Min RSSI Value*):** Minimum RSSI value that has to be received in the device to add the sending device (which has sent the broadcast) to its tags table (see *Broadcast* and *Tags*

Table messages in RTLS functional block). This value is also considered for RSSI port behaviors and rules (see *Define Behavior* and *Define Rule* messages in Autom functional block).

17. **Join network sleep period (*Join Network Sleep Period*):** If the device looses the network or it reports failure during the network start/join phase, period (in milliseconds) the device stays asleep before trying to join the Zigbee™ network again.
18. **Maximum join network attempts (*Max Join Network Attempts*):** The number of network join attempts the device performs, during network start/join phase, before it reports an internal failure. When these attempts finish the device goes to sleep if it has defined a join network sleep period or it starts again the network start/join phase. **The interval between two network join attempts is always set to 1000 ms.**
19. **Type of antenna (*Device Antenna Type*):** Type of antenna (hardware) used for transmitting and receiving data over the air. **For Sirius Quantum and Sirius RadIOn this parameter is a read-write parameter. For Sirius A and Sirius B/D, this parameter is a read-only parameter, that is, this value cannot be changed.**
20. **ZigBee™ network PAN ID (*ZB Network Pan Id*)**: 16-bit identification of the ZigBee™ network. This identification must be unique for all the nodes connected to the same ZigBee™ network.
21. **ZigBee™ extended PAN ID (*ZB Extended Pan Id*)**: 64-bit identification of ZigBee™ network. If zero value is set then the node will join to the first ZigBee™ network it detects on the air. This value must be unique for all the nodes connected to the same ZigBee™ network.
22. **ZigBee™ channel page (*ZB Channel Page*)**: Defines the modulation and transmission data rate used for the device. This parameter depends on the frequency band is being used:
    - **Frequency Band 2.4 GHz**: It will take always value 0 (O-QPSK modulation), that defines a 250kbps transmission data rate.
    - **Frequency Band 900 MHz (868 MHz/915 MHz):** It can take two different values:
        - Value 0: BPSK modulation and 20 kbps data rate.
        - Value 2: O-QPSK modulation and 100 kbps data rate.
23. **ZigBee™ channel mask (*ZB Channel Mask*)**: 32-bit parameter that defines the frequency band channels the node supports. This parameter depends on the frequency band is being used:
    - **Frequency Band 2.4 GHz**: There are 16 available channels in this frequency that fit with the ZigBee™ channels numbers 11 to 26. A node can support a single channel or several channels and the user have to select them in the next way:

      For example, if a node is desired to work only in channel number 17:
      0_{31} 0_{30} 0_{29} 0_{28} 0_{27} 0_{26} 0_{25} 0_{24} 0_{23} 0_{22} 0_{21} 0_{20} 0_{19} 0_{18} 1_{17} 0_{16} 0_{15} 0_{14} 0_{13} 0_{12} 0_{11} 0_{10} 0_{9} 0_{8} 0_{7} 0_{6} 0_{5} 0_{4} 0_{3} 0_{2} 0_{1} 0_{0} = 0x00020000

      Note: *If a node is desired to work in several channels at the same time, they can be selected setting their corresponding bit to 1.*

    - **Frequency Band 900 MHz (868 MHz/915 MHz):** There is only one available channel in this frequency band. The ZB Channel Mask will be set always to 0.
24. **ZigBee™ network static addressing (*ZB NWK Unique Addr*)**: This Boolean parameter defines if the static addressing mode is used or not for this node.
    - If "true", then static addressing is selected, and the user must define the network address node (*Node Id*) through the API of n-Core® or using the n-Core® Configuration Tool (see the n-Core® Configuration Tool Manual). The user have to take into account that two nodes cannot be set with the same node id in the same ZigBee™ Network. It is important to point out that if a node is defined with static addressing, then it will keep the same address (*Node id*) even if the device resets or turns off, and it turns on again. It is recommended to use this static addressing if the user wants to take full control over the nodes in the network.
    - If "false", then dynamic addressing is selected, and the n-Core® platform will selected automatically a free address inside that ZigBee™ *Network*. This address generally will change

every time the node leaves the network and joins again (for example, if the node resets or turns off and turns on again).

25. **ZigBee™ network predefined PAN ID (*ZB NWK Predefined Pan Id*):** This Boolean parameter indicates if the ZB *Network Pan Id* used for the node will be always the same. This parameter is especially important for the network coordinator, because if it is defined as "false", then the coordinator will select randomly the value for the parameter ZB *Network Pan Id* when creates a ZigBee™ network. For example, if the coordinator resets then it will create a ZigBee™ network with a different *ZB Network Pan Id* from that of the previous created ZigBee™ network and the nodes connected to the first ZigBee™ network will not be able to communicate with the coordinator. It is recommended to always set this value to "true" unless this is the desired behavior.

- **n-Core++**
  - *ncMgmtParamId type in "/mgmt/mgmtparamstypes.h"*
- **n-Core.C**
  - *ncMgmtParamId_t in "/mgmt/ncmgmt.h"*
- **n-Core.NET**
  - *ncMgmtNetParamId in "/nCoreMgmtNet/IncludenCoreMgmtNet.cs"*

In Table 2, the *Mgmt* parameters and their default values are shown.

Table 2. Default values for the configuration parameters of the *Management* functional block.

Parameter	Default value	Read-only
UID	-	Yes
Node Id	• Coordinator: 0 • No coordinator: 65533	-
Tx Power	• 2.4 GHz: +3 dBm • 900 MHz: +5 dBm	-
Neib Table Size	• A/B/D: 9 • Quantum/RadIOn: 51	Yes
Max Children Amount	• A/B/D: 8 • Quantum/RadIOn: 50	Yes
Max Children Router Amount	• A/B/D: 8 • Quantum/RadIOn: 50	Yes
Max Network Depth	10	Yes
Collecting Node	0	-
Alive Period	10000 ms	-
Hardware Type	-	Yes
App Device type	Router	-
Firmware Version	-	Yes
Sleep Period	0 ms	-
Awake Period	0 ms	-
Broadcast Period	0 ms	-
Min RSSI Value	-100 dBm	-
Join Network Sleep Period	0 ms	-
Max Join Network Attempts	4	-
Device Antenna Type	Internal	• A/B/D: Yes • Quantum/RadIOn: -
ZB NWK Pan Id[2]	0x0123 (Hex)	-
ZB Extended Pan Id[2]	0x0011223344556677 (Hex)	-
ZB Channel Page	• 2.4 GHz: 0 • 900 MHz: 2	-
ZB Channel Mask	0x00010000 (Hex)	-
ZB NWK Unique Addr	True	-
ZB NWK Predefined Pan Id	True	-

[2] This value may vary depending on the default configuration of the n-Core® devices.

2.6.6.2 *Management command request/response frames*

The command request and command response frames that exists in this block are the next ones:

1. **Obtaining a Management configuration parameter of a node (*GetParam*)**: it allows obtaining the value of a configuration parameter of a node connected to the ZigBee™ network.
   - **n-Core++:**
     - *ncMgmt::GetParam() method in "/mgmt/mgmt.h"*
     - *ncMgmtGetParamCommandResponseEvent class*
   - **n-Core.C:**
     - *ncMgmtGetParam() and ncMgmtDispatcherGetParam() functions in "/mgmt/ncmgmt.h"*
     - *ncMgmtGetParamCommandResponseEvent_t struct*
   - **n-Core.NET:**
     - *ncMgmtNetGetParam() method in "/nCoreMgmtNet/IncludenCoreMgmtNet.cs"*
     - *ncMgmtNetGetParamCommandResponseEvent class*

2. **Setting or modifying a Management configuration parameter of a node (*SetParam*)**: this command allows the user to modify the value of a configuration parameter of a node connected to the ZigBee™ network.
   - **n-Core++:**
     - *ncMgmt::SetParam() method in "/mgmt/mgmt.h"*
     - *ncMgmtSetParamCommandResponseEvent class*
   - **n-Core.C:**
     - *ncMgmtSetParam() and ncMgmtDispatcherSetParam() functions in "/mgmt/ncmgmt.h"*
     - *ncMgmtSetParamCommandResponseEvent_t struct*
   - **n-Core.NET:**
     - *ncMgmtNetSetParam() method in "/nCoreMgmtNet/IncludenCoreMgmtNet.cs"*
     - *ncMgmtNetSetParamCommandResponseEvent class*

3. **Ping to a node (*Ping*)**: it allows sending a *ping* to a node to check if the node is connected to the ZigBee™ network.
   - **n-Core++:**
     - *ncMgmt::Ping() method in "/mgmt/mgmt.h"*
     - *ncMgmtPingCommandResponseEvent class*
   - **n-Core.C:**
     - *ncMgmtPing() and ncMgmtDispatcherPing() functions in "/mgmt/ncmgmt.h"*
     - *ncMgmtPingCommandResponseEvent_t struct*
   - **n-Core.NET:**
     - *ncMgmtNetPing() method in "/nCoreMgmtNet/IncludenCoreMgmtNet.cs"*
     - *ncMgmtNetPingCommandResponseEvent class*

4. **Reset a node (*ResetNode*)**: it allows resetting a node. The node can be reset in two different ways:
   - <u>Node reset</u>: the node will keep all the values of that parameters stored in the EEPROM (there are several parameters that are stored automatically in the EEPROM depending on the working mode of the node).
   - <u>Node reset to default values</u>: it will modify all its parameters and they will be established to the default values.
   - **n-Core++:**
     - *ncMgmt::ResetNode() method in "/mgmt/mgmt.h"*
     - *ncMgmtResetNodeCommandResponseEvent class*

- **n-Core.C:**
  - *ncMgmtResetNode() and ncMgmtDispatcherResetNode() functions in "/mgmt/ncmgmt.h"*
  - *ncMgmtResetNodeCommandResponseEvent_t struct*
- **n-Core.NET:**
  - *ncMgmtNetResetNode() method in "/nCoreMgmtNet/IncludenCoreMgmtNet.cs"*
  - *ncMgmtNetResetNodeCommandResponseEvent class*

5. **Reset the Zigbee™ network (*ResetNetwork*)**: it allows forcing a node to leave the ZigBee™ network. It can be selected the time that will be expended before leaving the network and if the node will join or not the network again.
   - **n-Core++:**
     - *ncMgmt::ResetNetwok() method in "/mgmt/mgmt.h"*
     - *ncMgmtResetNetworkCommandResponseEvent class*
   - **n-Core.C:**
     - *ncMgmtResetNetwork() and ncMgmtDispatcherResetNetwork() functions in "/mgmt/ncmgmt.h"*
     - *ncMgmtResetNetworkCommandResponseEvent_t struct*
   - **n-Core.NET:**
     - *ncMgmtNetResetNetwork() method in "/nCoreMgmtNet/IncludenCoreMgmtNet.cs"*
     - *ncMgmtNetResetNetworkCommandResponseEvent class*

6. **Getting the neighbors table of a node connected to the ZigBee™ Network (*GetNeighborTable*)**: it allows obtaining the address, the LQI (Link Quality Indicator), the RSSI (Received Signal Strength Indication) and the relationship of its neighbors nodes.
   - **n-Core++:**
     - *ncMgmt::GetNeighborTable() method in "/mgmt/mgmt.h"*
     - *ncMgmtGetNeighborTableCommandResponseEvent class*
   - **n-Core.C:**
     - *ncMgmtGetNeighbors() and ncMgmtDispatcherGetNeighbors() functions in "/mgmt/ncmgmt.h"*
     - *ncMgmtGetNeighborsCommandResponseEvent_t struct*
   - **n-Core.NET:**
     - *ncMgmtNetGetNeighbors() method in "/nCoreMgmtNet/IncludenCoreMgmtNet.cs"*
     - *ncMgmtNetGetNeighborsCommandResponseEvent class*

7. **Getting the children table of a node connected to the ZigBee™ Network (*GetChildrenTable*)**: it allows obtaining the address of its children nodes.
   - **n-Core++:**
     - *ncMgmt::GetChildrenTable() method in "/mgmt/mgmt.h"*
     - *ncMgmtGetChildrenTableCommandResponseEvent class*
   - **n-Core.C:**
     - *ncMgmtGetChildren() and ncMgmtDispatcherChildren() functions in "/mgmt/ncmgmt.h"*
     - *ncMgmtGetChildrenCommandResponseEvent_t struct*
   - **n-Core.NET:**
     - *ncMgmtNetGetChildren() method in "/nCoreMgmtNet/IncludenCoreMgmtNet.cs"*
     - *ncMgmtNetGetChildrenCommandResponseEvent class*

2.6.6.3 Management message frames

Currently there is no message frame in the Management functional block.

IMPORTANT NOTE: Node Alive message is not anymore here in Mgmt functional block, as it has been moved to Kernel block since n-Core® 2.0.

2.6.7 Autom Functional Block

The **Automation Functional Block (*Autom*)** is composed by all the functionalities that allow the user to configure the *Sirius* devices to interact with different environment elements. Basically this block allows the user to obtain environment data (*Sensing*) and act over environment elements (*Acting*).

- **n-Core++**
  - *"/autom/autom.h"*
  - *ncore++_autom.dll* / l*ilibncore++_autom.so*
- **n-Core.C**
  - *"/autom/ncautom.h"*
  - *ncore_autom.dll / libncore_autom.so*
- **n-Core.NET**
  - *"/nCoreAutomNet"*
  - *ncorenet_autom.dll*

2.6.7.1 Sensors and Actuators

The *Sirius* devices include different input and output ports where two main different groups of elements can be connected:

1. **Sensor**: they can obtain/read data from the environment. There is a wide range of sensors: temperature sensors, presence detectors, ambient light sensors, humidity sensors, etc.
2. **Actuators**: they allow modifying and acting over environment elements. Example of actuators are electrical switches and relays, lights, alarms and sirens, door locks, etc.

The number of available ports of the Sirius devices vary depending on the Sirius model (A, B, D, Quantum or RadIOn), but all of them can be classified into the next type of ports:

1. **GPO (*General Purpose Output*):** Specific hardware pin defined as output data.
2. **GPI (*General Purpose Input*):** Specific hardware pin defined as input data.
3. **I^{2}C (*Inter-Integrated Circuit*):** Serial data communication bus. It can be used for reading or writing data depending on the device connected to it.
4. **PWM (*Pulse-Width Modulation*):** It allows producing a square-wave output signal whose frequency and duty-cycle can be varied.
5. **ADC (*Analog-to-Digital Converter*):** It converts a continuous (analog) signal (for example, voltage or current) into a discrete (digital) one. This port is used as input data.
6. **IRQ (*Interrupt ReQuest*):** Hardware request that interrupts the normal firmware flow in a device and executes the programmed action for that interrupt. This interrupt is used as input data for detecting impromptu inputs (for example, pressing a button in a Sirius B or a Sirius Quantum).
7. **BAT (*Battery*):** This port informs about the battery charge level of a Sirius device.
8. **EEPROM (*Electrically Erasable Programmable Read-Only Memory*):** It allows the user to store his/her own data into the EEPROM memory of the device. The user can access to the stored data later.
9. **RSSI (*Received Signal Strength Indication*):** This port informs about the RSSI value of the node with regard to another node or with regard to a group of nodes.

Note: Please, consult the datasheets and manuals of the n-Core® Sirius devices in order to know more about the number of ports available in each of them. In addition, see Annex C for detailed information about the relationship between pinout and port addresses in n-Core® Sirius devices.

In this regard, the n-Core® API provides different classes and types for representing these input and output ports in order to interact with the sensors and actuators connected to the Sirius devices.

- **n-Core++**
  - *ncAutomDevicePort and derived classes in "/autom/sensors/sensors.h"*
  - *ncAutomGPOPort class in "/autom/sensors/gpoport.h"*
  - *ncAutomGPIPort class in "/autom/sensors/gpiport.h"*
  - *ncAutomI2CPort class in "/autom/sensors/i2cport.h"*
  - *ncAutomPWMPort class in "/autom/sensors/pwmport.h"*
  - *ncAutomADCPort class in "/autom/sensors/adcport.h"*
  - *ncAutomIRQPort class in "/autom/sensors/irqport.h"*
  - *ncAutomBATPort class in "/autom/sensors/batport.h"*
  - *ncAutomEEPROMPort class in "/autom/sensors/eepromport.h"*
  - *ncAutomRSSIPort class in "/autom/sensors/rssiport.h"*
- **n-Core.C**
  - *ncAutomPort_t in "/autom/ncautom.h". See also:*
    - *ncAutomGPIParams_t in "/autom/ncautom.h"*
    - *ncAutomI2CParams_t in "/autom/ncautom.h"*
    - *ncAutomADCParams_t in "/autom/ncautom.h"*
    - *ncAutomPWMParams_t in "/autom/ncautom.h"*
    - *ncAutomIRQParams_t in "/autom/ncautom.h"*
    - *ncAutomRSSIParams_t in "/autom/ncautom.h"*
- **n-Core.NET**
  - *ncAutomNetDevicePort and derived classes in "/nCoreAutomNet/Ports"*
  - *ncAutomNetGPOPort class in "/nCoreAutomNet/Ports"*
  - *ncAutomNetGPIPort class in "/nCoreAutomNet/Ports"*
  - *ncAutomNetI2CPort class in "/nCoreAutomNet/Ports"*
  - *ncAutomNetADCPort class in "/nCoreAutomNet/Ports"*
  - *ncAutomNetPWMPort class in "/nCoreAutomNet/Ports"*
  - *ncAutomNetIRQPort class in "/nCoreAutomNet/Ports"*
  - *ncAutomNetBATPort class in "/nCoreAutomNet/Ports"*
  - *ncAutomNetEEPROMPort class in "/nCoreAutomNet/Ports"*
  - *ncAutomNetRSSIPort class in "/nCoreAutomNet/Ports"*

These ports are then used in *behaviors* and *rules* definitions (see *Behaviors and Rules* in this functional block) that can be sent to the Sirius devices in order to configure dynamically them for gathering input data from sensors as well as varying the output of actuators according to the collected sensing data.

2.6.7.2 Behaviors and Rules

The Automation functional block is based in two essential concepts intended for configuring the way that nodes interact with sensors and actuators in a remote and dynamic manner: ***behaviors*** and ***rules***.

1. **Behavior**: Action to read or write data from/to a certain port (input or output) of a device. Examples:
   - The user can configure a Sirius A device to read the value of the channel number 2 of its ADC every 5000 ms and to send only the obtained value to its collecting node if that value exceeds a certain threshold.
   - A device can be configured to detect a low-level interrupt in a specific IRQ port (for example, IRQ6) and to send a message to its collecting node to inform about this interrupt.
   - The user can set the GPO of a device with a high value to activate a certain actuator (for example, an alarm).

- A device can be configured to write data periodically in a specific address to its I²C bus and immediately read a certain amount of bytes from this I²C bus. This behavior could be used to obtain the temperature measured by a sensor connected to the I²C bus of the *Sirius* device.

- **n-Core++:**
  - *ncAutomBehaviorDefinition class in "/autom/behaviordefinition.h"*
- **n-Core.C:**
  - *ncAutomBehaviorDefinition_t struct in "/autom/ncautom.h"*
- **n-Core.NET:**
  - *ncAutomNetBehaviorDefinition class in "/nCoreAutomNet/Behaviors/ncAutomNetBehaviorDefinition.cs"*

2. **Rule**: it can be defined as the linking of two behaviors. When the first behavior of the rule is executed successfully then the second behavior will be executed. The second behavior can be defined in the node where it has been defined the first one or it can be defined in another node of the ZigBee™ network. Examples:
   - The user can define a node to check periodically its ADC port (first behavior) and if the read value is under certain threshold value then a GPO of other node will be activated (second behavior). The first node could have connected a smoke sensor to the ADC and the second node could have connected a fire alarm through a relay.
   - A node can be configured to read periodically the ambient light through a sensor connected to the I²C (first behavior). If the measured value is under a threshold value then a group of lights connected to the device can be activated (second behavior). This is the way to create applications for energy control and for automatic lighting.

     Notes:
     - *The first behavior of the rule always acts in the node where the rule has been defined in.*
     - *If both behaviors of a rule are defined to be executed in the node where the rule has been defined in, then both behaviors will be stored in this node at the moment the rule is defined.*
     - *On the other hand, if the second behavior belongs to a different node and all the rule definition is included when the rule is defined, then every time the first behavior of the rule is executed successfully the node where the rule is defined will send through the ZigBee™ network a definition behavior request command to the node where the second behavior has to be executed.*

   - **n-Core++:**
     - *ncAutomRuleDefinition class in "/autom/ruledefinition.h"*
   - **n-Core.C:**
     - *ncAutomRuleDefinition_t struct in "/autom/ncautom.h"*
   - **n-Core.NET:**
     - *ncAutomNetRuleDefinition class in "/nCoreAutomNet/Rules/ncAutomNetRuleDefinition.cs"*

2.6.7.3 Configuration parameters of the Automation functional block

Currently there is no parameter in the Automation functional block.

2.6.7.4 Automation command request/response frames

The command request/response frames included in this block are the next ones:

1. **Obtaining an Automation configuration parameter of a node[3] (*GetParam*)**: it allows obtaining the value of an *Autom* configuration parameter of a node connected to the ZigBee™ network.
    - **n-Core++:**
        - *Not available (no Autom parameter defined by the moment)*
    - **n-Core.C:**
        - *Not available (no Autom parameter defined by the moment)*
    - **n-Core.NET:**
        - *Not available (no Autom parameter defined by the moment)*

2. **Setting or modifying an Automation configuration parameter of a node**3 **(*SetParam*)**: this command allows the user to modify the value of an *Autom* configuration parameter of a node connected to the ZigBee™ network.
    - **n-Core++:**
        - *Not available (no Autom parameter defined by the moment)*
    - **n-Core.C:**
        - *Not available (no Autom parameter defined by the moment)*
    - **n-Core.NET:**
        - *Not available (no Autom parameter defined by the moment)*

3. **Defining a behavior in a node (*DefineBehavior*):** it allows the user defining a behavior in a specific node. When a behavior is defined, several parameters must be defined. Next the most important ones are explaining:
    - Behavior Identification (*BehaviorId*): Two different values can be used in this field:
        - Zero Value (*NEW*): Indicates that the user wants to define a new behavior in the node. This behavior will be stored in a free position of the node. The response will inform about the position (which *BehaviorId*) where the behavior has been stored.
        - Value[4] between 1 and 10 (both included): If a non-zero value is selected then the behavior will be stored in that position. If a behavior was stored previously in that position, then that behavior will be overwritten.
    - Behavior period (*Period*): Period used for reading and/or writing data:
        - If the period is equal to zero then the behavior (reading and/or writing data) will be executed only once, immediately after defining it. Must be pointed out that if the period of a behavior is zero and the behavior is not persistent, the behavior will be deleted from the node when it has been executed, releasing the position the behavior was occupying in the node.
        - If a period higher than zero is defined then reading/writing data will be executed every time that amount of time is elapsed.
    - Device port (*DevicePort*): parameters related to the device port involved.
        - Port type (*DevicePortType*): It will be one of the ports explained above:
            - GPI
            - GPO
            - I^{2}C
            - PWM
            - ADC
            - IRQ
            - BAT
            - EEPROM

[3] Currently there is not defined any configuration parameter in this functional block.

[4] The maximum number of behaviors that can be defined in a Sirius device is 10.

- RSSI
- Port Identification[5] (*PortId*): This parameter is used for distinguishing a port from another when in a device has several port of the same type. For example, if a device has two I^{2}C buses or two ADC ports.

  Note: *do not confuse this parameter with the PortAddress parameter.*

- Port Address (*PortAddress*): This parameter indicates the address that will be used for reading or writing data in a specific port id of a certain port type. Depending on the port type, this parameter will have a different meaning. *For example, in an ADC it indicates the number of the ADC channel used; in the GPO/GPI it indicates the hardware pin where the port is connected; in the I^{2}C bus it indicates the I^{2}C bus address, etc.*

  Note: *See Annex C for detailed information about the relationship between port addresses and pinout in n-Core® Sirius devices.*

- Variable parameters according to the type of port. Sometimes it can be necessary to add some specific parameters according to the type of port of the behavior:
  - GPI:
    - *Internal Pull-up:* It allows the user to activate or deactivate the internal pull-up resistance that every GPI of the device contains.

      Note: *For the Sirius A devices this internal pull-up must be always deactivated.*
  - I^{2}C:
    - *I^{2}C Rate:* Its available values are:
      - 62.5 Kbps
      - 125 Kbps
      - 250 Kbps
    - *I^{2}C Internal Address:* There are integrated circuits that allow accessing certain internal memory positions to be read or modified through the I2C.
  - PWM:
    - *Frequency*.
    - *Duty cycle*.
  - ADC:
    - *Sample Rate:* Available values (*sps = samples per second*):
      - 4800 sps
      - 9600 sps
      - 19200 sps
      - 39 Ksps
      - 77 Ksps
    - *Resolution* (*BitsPerSample*): Available values:
      - 8 bits per sample
      - 10 bits per sample
  - IRQ:

[5] This parameter is always 0 in the current n-Core® Sirius devices.

- *Interrupt condition type* (*IRQConditionType*): It indicates the digital input type that will generate an interruption into the selected hardware pin. The possible values are:
  - LOW_LEVEL
  - HIGH_LEVEL[6]
  - ANY_EDGE
  - FALLING_EDGE
  - RISING_EDGE
- *Guard time*: minimum interval between two consecutive IRQs (in milliseconds).

- RSSI
  - *Initial node identification (Initial Node Id)*: it defines the address of the node (or the lowest node of a group of nodes) regarding which the RSSI will be measured.
  - *Final node identification (Final Node Id)*: it defines the address of the node (or the highest node of a group of nodes) regarding which the RSSI will be measured. This parameter must be equal to the *Initial Node Id* if you only want to measure the RSSI regarding a unique node and not regarding a group of nodes.

- Sequence of bytes to be written to the port (*BytesToBeWritten*): This parameter indicates the sequence of bytes that will be written to a specific port.

  Note: *this field must be empty if the port where the behavior is defined is read-only (input) or if the user does not want to write anything to the port or even if the port is an output port but it is not possible to write a sequence of bytes to it (for example, a PWM port).*

- Number of bytes to read from the port (*BytesToBeReadNumber*): This parameter indicates the total number of bytes that will be read from the port. The user can select a subset of bytes (*token*) from that total read bytes in order to obtain only the data that the user is interested in.

  Note: *this field must be zero if the port where the behavior is defined is write-only (output) or if the user does not want to read anything from the port or even if the port is an input port but it is impossible to read detailed data from it (for example, an IRQ port).*

- Initial index of the subset of bytes (token) selected inside the total read bytes (*ReadTokenStartIndex*).
- Type of token to be read (*ReadTokenType*): for example, if the read token is an unsigned 16-bit integer, a signed 32-bit integer, a buffer of bytes, a Boolean value, etc.

  Note: *see Parameter Types section in the Common functional block.*

- If the read token is in MSB (Most Significant Byte) order or not (*MSBreadToken*).

  Note: *these last five parameters (BytesToBeWritten, BytesToBeReadNumber, ReadTokenStartIndex, ReadTokenType and MSBreadToken) are better explained with an example:*

  Let's suppose there is a temperature sensor connected to the I²C bus of a specific node. According to its specifications, to obtain the temperature it is necessary to send some data to the sensor (0xAA 0xBB) and then the sensor will response with

[6] Currently this kind of interrupt is not available.

some data (0xAA 0xXX 0xXX) where the second and third bytes form a 16-bit integer that represents the ambient temperature in MSB format. So in this case we will have the next values for these parameters: BytesToBeWritten = “0xAA 0xBB”, BytesToBeReadNumber = “3”, ReadTokenStartIndex = “1” (because the first byte is the byte 0), ReadTokenType = “UINT16” and MSBreadToken = “true”.

- Condition used to compare read data (*Condition*): The read data can be compared with a certain condition in such way that the node will only send the data message frames to the source/collecting node in case this condition is achieved. The user can establish the case in which the node will send the data messages using the next parameters (*ConditionType* and *Reference*):
    - Type of the condition for a behavior step (*ConditionType*):
        - ALWAYS: the data read is not compared with any reference value, so messages will be always sent.
        - NEVER: the data messages from this behavior will never be sent. This case can be useful for those write-only behaviors when the API is not necessary to be informed (autonomous systems) or when the behavior is included in a rule and the user only wants to receive the data messages from the second behavior or from the complete rule.
        - IF_EQUALS_TO: the node will only send data messages if the read value is equals to the reference value.
        - IF_NOT_EQUALS_TO: the node will only send data messages if the read value is different from the reference value.
        - IF_HIGHER_OR_EQUALS_TO: the node will only send data messages if the read value is equals to or higher than the reference value.
        - IF_LOWER_OR_EQUALS_TO: the node will only send data messages if the read value is equals to or lower than the reference value.
    - Reference value (*Reference*): reference value with the same type as the (*ReadTokenType*). The value of the read token is compared with the reference value using the operator given by the *ConditionType*.
      Note: *see Parameter Types section in the Common functional block.*

- *Persistence*: Boolean parameter that indicates if the behavior will be persistent or not:
    - *Persistent*: The behavior will not be deleted from the node until the user deletes it explicitly. The behavior will be kept defined in the node even if the node is reset or turned off.
    - *Non-persistent*: the behavior will be deleted from the node if the device is reset or turned off.
- Send message frames to the source node (*NotifySrcNode*): Boolean parameter that indicates if the message frames that are sent when the behavior has been executed successfully (see *DefineBehavior* messages in this functional block) will be sent or not to the node from which this behavior has been defined. For example, if a behavior in the node number 3 is remotely configured from the API through the node number 5 connected to the computer, then the message frames will be sent from node number 3 to node number 5 if this parameter has been defined to “true”.
  Note: *all the devices can receive these messages but only the coordinator or the collecting nodes are able to forward these received messages through the serial port (USB or RS-232).*

- Send message frames to the collecting node (*NotifyCollector*): Boolean parameter that indicates if the message frames that are sent when the behavior has been executed successfully (see *DefineBehavior* messages in this functional block) will be sent or not to the collecting node of this device.

> ***Note:*** *if the collecting node and the source node are the same one, then the message frames will be only sent once even if both parameters (NotifySrcNode and NotifyCollector) are defined as "true".*

- **n-Core++:**
  - *ncAutom::DefineBehavior() method in "/autom/autom.h"*
  - *ncAutomDefineBehaviorCommandResponseEvent class*
- **n-Core.C:**
  - *ncAutomDefineBehavior() and ncAutomDispatcherDefineBehavior() functions in "/autom/ncautom.h"*
  - *ncAutomDefineBehaviorCommandResponseEvent_t struct*
- **n-Core.NET:**
  - *ncAutomNetDefineBehavior() method in "/nCoreAutomNet/IncludenCoreAutomNet.cs"*
  - *ncAutomNetDefineBehaviorCommandResponseEvent class*

4. **Removing a behavior from a node (*ResetBehavior*)**: it allows the user deleting one behavior or several ones defined in a node. By means of this command request the user can delete one behavior or all the behaviors of one type (persistent, non-persistent or both) defined in a node.
   - **n-Core++:**
     - *ncAutom::ResetBehavior() method in "/autom/autom.h"*
     - *ncAutomResetBehaviorCommandResponseEvent class*
   - **n-Core.C:**
     - *ncAutomResetBehavior() and ncAutomDispatcherResetBehavior() functions in "/autom/ncautom.h"*
     - *ncAutomResetBehaviorCommandResponseEvent_t struct*
   - **n-Core.NET:**
     - *ncAutomNetResetBehavior() method in "/nCoreAutomNet/IncludenCoreAutomNet.cs"*
     - *ncAutomNetResetBehaviorCommandResponseEvent class*

5. **Obtaining the list of behaviors defined in a node (*GetActiveBehaviors*)**: it allows obtaining the behaviors a node has defined in. Through this command request the user can obtain the identifications of the behaviors defined in a node (persistent, non-persistent or both).
   - **n-Core++:**
     - *ncAutom::GetActiveBehaviors() method in "/autom/autom.h"*
     - *ncAutomGetActiveBehaviorsCommandResponseEvent class*
   - **n-Core.C:**
     - *ncAutomGetActiveBehaviors() and ncAutomDispatcherGetActiveBehaviors() functions in "/autom/ncautom.h"*
     - *ncAutomGetActiveBehaviorsCommandResponseEvent_t struct*
   - **n-Core.NET:**
     - *ncAutomNetGetActiveBehaviors() method in "/nCoreAutomNet/IncludenCoreAutomNet.cs"*
     - *ncAutomNetGetActiveBehaviorsCommandResponseEvent class*

6. **Obtaining the definition of a behavior active and defined in a node (*GetActiveBehaviorDefinition*):** it allows the user obtaining all the parameters of a behavior defined in a node.
   - **n-Core++:**

    - *ncAutom::GetActiveBehaviorDefinition() method in "/autom/autom.h"*
    - *ncAutomGetActiveBehaviorDefinitionCommandResponseEvent class*
  - **n-Core.C:**
    - *ncAutomGetActiveBehaviorDefinition() and ncAutomDispatcherGetActiveBehaviorDefinition() functions in "/autom/ncautom.h"*
    - *ncAutomGetActiveBehaviorDefinitionCommandResponseEvent_t struct*
  - **n-Core.NET:**
    - *ncAutomNetGetActiveBehaviorDefinition() method in "/nCoreAutomNet/IncludenCoreAutomNet.cs"*
    - *ncAutomNetGetActiveBehaviorDefinitionCommandResponseEvent class*

7. **Defining a rule in a node (*DefineRule*):** it allows the user to define a rule in a certain node. When a rule is defined, several parameters need to be defined. In addition to these parameters, when a rule is defined it is necessary to specify the two behaviors which the rule will be composed by. These behaviors can be totally two new ones or they can be already defined previously in the nodes. In this last case, the rule can concatenate the behaviors already defined:
   - If any of the two behaviors has not been defined previously in the node or nodes that take part in the rule, then it is necessary to include all the behavior definition into the rule definition.
   - If any of the two behaviors has been defined previously in the node or nodes that take part in the rule, then it is only necessary to include its behavior identification into the rule definition.

   Next, the most important parameters in the rule definition are described:
   - Rule identification (*RuleId*): Two different values can be used in this field:
     - Zero Value (*NEW*): Indicates that the user wants to define a new rule in the node. This rule will be stored in a free position of the node. The response will inform about the position where the rule (which *RuleId*) has been stored.
     - Value[7] between 1 and 5 (both included): If a non zero value is selected then the rule will be stored in that position. If a rule was stored previously in that position, then that rule will be overwritten.
       Note: *It is strongly recommended to use zero value in this field because overwriting a previous defined rule can cause several failures in the behaviors of that rule.*

   - Second node identification (*SecondNodeId*): Address of the node where the second behavior of the rule is defined and executed.
     Note: *There is no need to have a parameter representing the first node id, as the first behavior is always defined where the rule is defined.*

   - Existence of the first behavior (*FirstBehaviorExistence*): If the first behavior in the rule existed previously or not. That is, if the first behavior was previously defined in the node or nodes that take part in the rule or if the user is defining at the same time the rule is being defined.
   - Existence of the second behavior (*SecondBehaviorExistence*): If the second behavior in the rule existed previously or not. That is, if the second behavior was previously defined in the node or nodes that take part in the rule or if the user is defining at the same time the rule is being defined.
   - Send message frames to the source node (*NotifySrcNode*): Boolean parameter that indicates if the message frames that are sent when the rule has been executed successfully

[7] The maximum number of rules that can be defined in a Sirius device is 5.

(see *DefineRule* messages in this functional block) will be sent or not to the node from which this rule has been defined.

> ***Note:*** *all the devices can receive these messages but only the coordinator or the collecting nodes are able to forward these received messages through the serial port (USB or RS-232).*

- Send message frames to the collecting node (*NotifyCollector*): Boolean parameter that indicates if the message frames that are sent when the rule has been executed successfully (see *DefineRule* messages in this functional block) will be sent or not to the collecting node of this device.

  > ***Note:*** *if the collecting node and the source node are the same one, then the message frames will be only sent once even if both parameters (NotifySrcNode and NotifyCollector) are defined as "true".*

- Persistence[8]: Boolean parameter that indicates if the rule will be persistent or not:
  - *Persistent*: The rule will not be deleted from the node until the user deletes it explicitly. The rule will be kept defined in the node even if the node is reset or turned off.
    - If you want a rule to be persistent, it is necessary that the first behavior of the rule is also persistent.
    - For the second behavior of the rule:
      - If the second behavior of the rule belongs to the node where the rule is being defined, then it is necessary that this second behavior is also persistent.
      - If the second behavior belongs to a different node where the rule is being defined, then the second behavior can be persistent or non-persistent.
  - *Non-persistent*: the rule will be deleted from the node if the device is reset or turned off.
- Transferring data between the two behaviors (*DataTransferBetweenBehaviors*): This parameter indicates if the data read by the first behavior will be used as the data to be written by the second behavior. Among other restrictions, to carry out successfully the transferring data between the two behaviors it is strictly necessary that the first behavior must be a behavior that allows reading data from its port and the second one must be a behavior that allows writing data to its port.
- Identification of the first behavior (*FirstBehaviorId*).
- Complete definition, if any, of the first behavior (*FirstBehaviorDefinition*).
- Identification of the second behavior (*SecondBehaviorId*).
- Complete definition, if any, of the second behavior (*SecondBehaviorDefinition*).

- **n-Core++:**
  - *ncAutom::DefineRule() method in "/autom/autom.h"*
  - *ncAutomDefineRuleCommandResponseEvent class*
- **n-Core.C:**
  - *ncAutomDefineRule() and ncAutomDispatcherDefineRule() functions in "/autom/ncautom.h"*
  - *ncAutomDefineRuleCommandResponseEvent_t struct*
- **n-Core.NET:**

[8] Independently if the rule is persistent or not, if the second behavior of the rule is defined in a different node than the node where the rule is defined it is strongly recommended that this second behavior will be defined as no persistent to avoid inconsistencies in the second node.

        - *ncAutomNetDefineRule() method in "/nCoreAutomNet/IncludenCoreAutomNet.cs"*
        - *ncAutomNetDefineRuleCommandResponseEvent class*

8. **Removing a rule from a node (*ResetRule*)**: it allows deleting one rule or several ones defined in a certain node. By means of this command request the user can delete one rule or all the rules of one type (persistent, non-persistent or both) defined in a node.
    - When the rule is defined, if the two behaviors or any of them was a new behavior, this behavior or these ones defined as new ones will be deleted when the rule is deleted independently of its persistence.
    - When the rule is defined, if the two behaviors or any of them was already defined in the node before defining the rule, these behaviors will NOT be deleted when the rule is deleted and they will keep running in the node.

    - **n-Core++:**
        - *ncAutom::ResetRule() method in "/autom/autom.h"*
        - *ncAutomResetRuleCommandResponseEvent class*
    - **n-Core.C:**
        - *ncAutomResetRule() and ncAutomDispatcherResetRule() functions in "/autom/ncautom.h"*
        - *ncAutomResetRuleCommandResponseEvent_t struct*
    - **n-Core.NET:**
        - *ncAutomNetResetRule() method in "/nCoreAutomNet/IncludenCoreAutomNet.cs"*
        - *ncAutomNetResetRuleCommandResponseEvent class*

9. **Obtaining the list of rules defined in a node (*GetActiveRules*)**: it allows obtaining the rules that a node has defined in. Through this command request the user can obtain the identifications of the rules defined in a node (persistent, non-persistent or both).
    - **n-Core++:**
        - *ncAutom::GetActiveRules() method in "/autom/autom.h"*
        - *ncAutomGetActiveRulesCommandResponseEvent class*
    - **n-Core.C:**
        - *ncAutomGetActiveRules() and ncAutomDispatcherGetActiveRules() functions in "/autom/ncautom.h"*
        - *ncAutomGetActiveRulesCommandResponseEvent_t struct*
    - **n-Core.NET:**
        - *ncAutomNetGetActiveRules() method in "/nCoreAutomNet/IncludenCoreAutomNet.cs"*
        - *ncAutomNetGetActiveRulesCommandResponseEvent class*

10. **Obtaining the definition of a rule active and defined in a node (*GetActiveRuleDefinition*):** it allows the user to obtain all the parameters of a rule defined in a node.
    - **n-Core++:**
        - *ncAutom::GetActiveRuleDefinition() method in "/autom/autom.h"*
        - *ncAutomGetActiveRuleDefinitionCommandResponseEvent class*
    - **n-Core.C:**
        - *ncAutomGetActiveRuleDefinition() and ncAutomDispatcherGetActiveRuleDefinition() functions in "/autom/ncautom.h"*
        - *ncAutomGetActiveRuleDefinitionCommandResponseEvent_t struct*
    - **n-Core.NET:**
        - *ncAutomNetGetActiveRuleDefinition() method in "/nCoreAutomNet/IncludenCoreAutomNet.cs"*

- *ncAutomNetGetActiveRuleDefinitionCommandResponseEvent class*

2.6.7.5 Automation message frames

The message frames that exist in this block are:

1. **Message result of a behavior in a node (*DefineBehavior*)**: this message is generated when a behavior has been executed successfully and it is defined that this message has to be sent to the source node and/or to the collecting node. This message contains the status of the behavior execution, the behavior identification, the behavior complete definition, as well as the read data (if the behavior included data to be read).
   - **n-Core++:**
     - *ncAutom::RegisterDefineBehaviorMessageEventHandler() method in "/autom/autom.h"*
     - *ncAutom::UnregisterDefineBehaviorMessageEventHandler() method in "/autom/autom.h"*
     - *ncAutomDefineBehaviorMessageEvent class*
   - **n-Core.C:**
     - *ncAutomRegisterDefineBehaviorMessageEventHandler() function in "/autom/ncautom.h"*
     - *ncAutomUnregisterDefineBehaviorMessageEventHandler() function in "/autom/ncautom.h"*
     - *ncAutomRegisterDefineBehaviorMessageEventDispatcher() function in "/autom/ncautom.h"*
     - *ncAutomUnregisterDefineBehaviorMessageEventDispatcher() function in "/autom/ncautom.h"*
     - *ncAutomDefineBehaviorMessageEvent_t struct*
   - **n-Core.NET:**
     - *ncAutomNetRegisterDefineBehaviorMessageEventHandler() method in "/nCoreAutomNet/IncludenCoreAutomNet.cs"*
     - *ncAutomNetUnregisterDefineBehaviorMessageEventHandler() method in "/nCoreAutomNet/IncludenCoreAutomNet.cs"*
     - *ncAutomNetRegisterDefineBehaviorMessageEventDispatcher() method in "/nCoreAutomNet/IncludenCoreAutomNet.cs"*
     - *ncAutomNetUnregisterDefineBehaviorMessageEventDispatcher() method in "/nCoreAutomNet/IncludenCoreAutomNet.cs"*
     - *ncAutomNetDefineBehaviorMessageEvent class*

2. **Message result of a rule in a node (*DefineRule*)**: this message is created when a complete rule has been executed successfully (that is, the two behaviors) and if it has been defined that this message has to be sent to the source node and/or the collecting node. This message contains the status of the rule execution, the rule identification and the rule complete definition.
   - **n-Core++:**
     - *ncAutom::RegisterDefineRuleMessageEventHandler() method in "/autom/autom.h"*
     - *ncAutom::UnregisterDefineRuleMessageEventHandler() method in "/autom/autom.h"*
     - *ncAutomDefineRuleMessageEvent class*
   - **n-Core.C:**
     - *ncAutomRegisterDefineRuleMessageEventHandler() function in "/autom/ncautom.h"*
     - *ncAutomUnregisterDefineRulerMessageEventHandler() function in "/autom/ncautom.h"*

    - *ncAutomRegisterDefineRuleMessageEventDispatcher() function in "/autom/ncautom.h"*
    - *ncAutomUnregisterDefineRuleMessageEventDispatcher() function in "/autom/ncautom.h"*
    - *ncAutomDefineRuleMessageEvent_t struct*
- **n-Core.NET:**
    - *ncAutomNetRegisterDefineRuleMessageEventHandler() method in "/nCoreAutomNet/IncludenCoreAutomNet.cs"*
    - *ncAutomNetUnregisterDefineRuleMessageEventHandler() method in "/nCoreAutomNet/IncludenCoreAutomNet.cs"*
    - *ncAutomNetRegisterDefineRuleMessageEventDispatcher() method in "/nCoreAutomNet/IncludenCoreAutomNet.cs"*
    - *ncAutomNetUnregisterDefineRuleMessageEventDispatcher() method in "/nCoreAutomNet/IncludenCoreAutomNet.cs"*
    - *ncAutomNetDefineRuleMessageEvent class*

2.6.8 RTLS Functional Block

The **RTLS functional block (RTLS)** is composed by all the n-Core® functionalities that allows configuring and using all the features related to the **n-Core® locating engine**.

Note: the n-Core® locating engine is available only with the Complete n-Core® License.

- **n-Core++**
    - *"/rtls/rtls.h"*
    - *ncore++_rtls.dll / libncore++_rtls.so*
- **n-Core.C**
    - *"/rtls/ncrtls.h"*
    - *ncore_rtls.dll / libncore_rtls.so*
- **n-Core.NET**
    - *"/nCoreRtlsNet"*
    - *ncorenet_rtls.dll*

2.6.8.1 Nodes involved in the RTLS process

All the Sirius devices that are included into the RTLS infrastructure must have one of these roles:

1. **Coordinator**: it is the same as the network coordinator. By default all the routers connected to the ZigBee™ network will send its location information (*TagsTable* message frames) to the coordinator. There must be only one coordinator in every single ZigBee™ network. If a router has defined another node as collecting node different from the coordinator then the location information will only be received by that collecting node.
   The coordinator will also receive the information sent by the tags (*Broadcast* message frames), as a router, and it will be able to create the location information (*TagsTable* message frames) if it has defined a non-zero *Tags Table Period*.
   The coordinator can send *Broadcast* message frames if it has defined a *Broadcast Period* different from zero.
   This RTLS role matches with the "Coordinator" application device type (*App Device Type*) defined in the Management functional block.

2. **Router**: it will receive the information sent by the tags. This information is used to create the location data (*TagsTable* message frames) that is sent by the router to the coordinator or the collecting node.

The routers can send *Broadcast* message frames if they have defined a *Broadcast Period* different from zero.
This RTLS role matches with the “Router” application device type (*App Device Type*) defined in the Management functional block.

3. **Tag**: it sends a *Broadcast* message frame every certain period (*Broadcast Period*) of time that will be received by those routers that are inside its coverage radio.
   This RTLS role matches with the “Tag” application device type (*App Device Type*) defined in the Management functional block.

4. **Router mobile**[9]: it will detect near tags and it will send its location information (*TagsTable* message frames) periodically to the coordinator or to its collecting node as a fixed router. Nevertheless, these routers difference from the fixed ones because their coordinates are not predefined as in a fixed router. The locating algorithms used in the RTLS system will calculate their position according to the relative position regarding the rest of devices.
   This RTLS role matches with the “Router” application device type (*App Device Type*) defined in the Management functional block.

5. **Collector**: it collects location information sent by the routers. By default, the routers send their location information (*TagsTable* message frames) to the coordinator.
   The collecting nodes will also receive the information sent by the tags (*Broadcast* message frames), as a router, and they will be able to create the location information (*TagsTable* message frames) if they have defined a *Tags Table Period* different from zero.
   The collectors are able to send broadcast frames if they have defined a *Broadcast Period* different from zero.
   This RTLS role matches with the “Collecting node” application device type (*App Device Type*) defined in the Management functional block.

2.6.8.2 Distance Estimators, Weightings and Locaters

The n-Core® API provides ***Locater*** classes/types that represent different locating algorithms that can be used.

Locaters make use of ***Distance Estimators*** classes/types in order to estimate the distances existing among the different nodes in the network (i.e., *Readers/Routers* and *Tags*) according to the detection information sent from the *Readers* within *TagsTable* messages.

In order to average and make smoother the distance estimated among nodes during the successive cycles (*See StartLocation in this functional block*), *Locaters* make use also of ***Weightings***.

Note: *See “RTLS methods/functions for managing the locating engine” in this functional block.*

Currently these are the ***Locaters*** offered by the n-Core® API, along with their main specific parameters:

1. ***SignpostLocater***: This technique determines the closest Reader to each Tag.
   - **Damping cycles.** The minimal number of locating cycles necessary for a tag to change its position between a reader and another.

[9] Currently not all the RTLS algorithms of the API of the n-Core® support the use of router mobile devices.

2. ***SimpleFuzzyLocater:*** This is an exclusive technique developed by Nebusens. It calculates the position of each Tag by using a fuzzy interpolation of the fixed Readers positions that detect each Tag.
    - ***Spatial.*** Performs the Simple Fuzzy technique in 2D (ignoring the Z coordinate) or 3D.
    - ***Parameter t.*** It is the attraction of the Tags to the Readers. A higher *t* value implies a stronger attraction (a very high *t* value implies a behavior similar to the signpost technique). By default, this value is 2.5 (recommended value).
    - ***Cell.*** Size (in centimeters) of the cells used in the simple fuzzy algorithm to estimate the location of the tags. A higher size involves a more stable visual representation of the position of the tags, reducing the resolution of the output data.

3. ***TrilaterationLocater***: This technique calculates the position of each Tag using a *trilateration* algorithm.
    - ***Spatial.*** Performs the Trilateration technique in 2D (ignoring the Z coordinate) or 3D.

Among the common parameters for all *Locaters* we have:

1. ***Distance Threshold.*** *Discriminates Readers too far from a given Tag. That is, Readers that are further than this threshold with respect to a certain Tag are not considered as references when calculating the position of that Tag. This is useful to reduce computational complexity and to build access control systems. It must be a non-negative number (finite or infinite). By default, it is infinity (to consider all readers that detect a certain Tag).*

- **n-Core++**
    - *ncRtlsLocater class in "/rtls/locater/locater.h"*
    - *ncRtlsSignpostLocater class in "/rtls/locater/signpostlocater.h"*
    - *ncRtlsSimpleFuzzyLocater class in "/rtls/locater/simplefuyzzylocater.h"*
    - *ncRtlsTrilaterationLocater class in "/rtls/locater/trilaterationlocater.h"*
- **n-Core.C**
    - *ncRtlsLocater_t in "/rtls/ncrtls.h"*
    - *ncRtlsSignpostLocater_t struct in "/rtls/ncrtls.h"*
    - *ncRtlsSimpleFuzzyLocater_t struct in "/rtls/ncrtls.h"*
    - *ncRtlsTrilaterationLocater_t struct in "/rtls/ncrtls.h"*
- **n-Core.NET**
    - *ncRtlsNetLocater class in "/nCoreRtlsNet/Locater"*
    - *ncRtlsNetSignPostLocater class in "/nCoreRtlsNet/Locater"*
    - *ncRtlsNetSimpleFuzzyLocater class in "/nCoreRtlsNet/Locater"*
    - *ncRtlsNetTrilaterationLocater class in "/nCoreRtlsNet/Locater"*

DistanceEstimator classes/types are used by *Locaters* for estimating distances between Tags and Readers from the detection values (RSSI, transmission power of the Tags, LQI, etc.).

Currently these are the ***DistanceEstimators*** offered by the n-Core® API, along with their main specific parameters

1. ***Mean Large Scale Path Loss.*** This distance estimator calculates distances from RSSI and the transmission power of Tags according to a radio propagation model of large-scale path-loss, as in the next formula:

$$\boldsymbol{d = d_o\left(10^{P_r(d)-P_r(d_o)}\right)^{-1/10n}}$$

where $P_r(d)$ is the RSSI between the tag and the reader.

- **Received Power.** Received power, $\mathbf{P_r}(\mathbf{d_o})$, at the *Initial Distance* (in dBm).
- **Initial Distance.** Certain distance, d_o, whose received power is equals to the *Received Power*.
- **Parameter n.** Path loss exponent, n, typically n**=2.0** in the free space (but higher in indoor environments).

2. ***RSSI and LQI Linear.*** It performs an estimation in this way:

$$d = RSSI_{coeff} \cdot RSSI + LQI_{coeff} \cdot LQI + offset$$

As $RSSI$ is usually negative and LQI is always a positive value, it is recommended to use a negative $RSSI_{coeff}$ value

- **n-Core++**
  - *ncRtlsDistanceEstimator class in "/rtls/locater/distanceestimator.h"*
  - *ncRtlsMeanLargeScalePathLossDistanceEstimator class in "/rtls/locater/meanlargescalepathlossdistanceestimator.h"*
  - *ncRtlsRSSIandLQILinearDistanceEstimator class in "/rtls/locater/rssiandlqilineardistanceestimator.h"*
- **n-Core.C**
  - *ncRtlsDistanceEstimator_t struct in "/rtls/ncrtls.h"*
  - *ncRtlsMeanLargeScalePathLossDistanceEstimator_t struct in "/rtls/ncrtls.h"*
  - *ncRtlsRSSIandLQILinearDistanceEstimator_t struct in "/rtls/ncrtls.h"*
- **n-Core.NET**
  - *ncRtlsNetDistanceEstimator class in "/nCoreRtlsNet/Locater"*
  - *ncRtlsNetMeanLargeScalePathLossDistanceEstimator class in "/nCoreRtlsNet/Locater"*
  - *ncRtlsNetRSSIandLQILinearDistanceEstimator class in "/nCoreRtlsNet/Locater"*

Weighting classes/types act as a low-pass filter that average the distance values obtained by the chosen *DistanceEstimator*. These averaged distance values are passed to the *Locater* to perform the locating technique.

Among the common parameters for all *Weightings* we have:

1. ***Cycles/Size*:** The number of elements in the weighting data. That is, the number of recent locating cycles that are averaged to smooth the estimated distances among nodes. Therefore, if cycles are 3, the weighting will average the 3 newest cycles.
2. ***WorstDistance*:** The value used for a Reader that has not detected a Tag in some cycle. The special value "Not-A-Number" (NAN) means that the mean distance will not be changed.

Currently these are the ***Weightings*** offered by the n-Core® API, along with their main specific parameters.

1. ***Uniform.*** Uniform weighting (all weightings are the same).
2. ***Geometric.*** Geometric weighting. The newest element (the first in the data) has a value *factor* times greater than the previous, and so for all elements
   - ***Factor.*** It is used to give the last data more weight. For example, if *factor* is 1.1, each cycle weights 1.1 times more than the older one.
3. ***Arithmetic.*** The newest element (the first in the data) has a value equal to the previous plus an *echelon*, and so for all elements.

- ***Echelon***. It is used to give the last data more weight. For example, if *echelon* is 0.2, each cycle weights 1.2 times more than the older one.

- **n-Core++**
  - *ncRtlsWeighting and derived classes in "/rtls/locater/weighting.h"*
- **n-Core.C**
  - *ncRtlsWeighting_t, ncRtlsUniformWeightingParams_t, ncRtlsGeometricWeightingParams_t and ncRtlsArithmeticWeightingParams_t structs in "/rtls/ncrtls.h*
- **n-Core.NET**
  - *ncRtlsNetWeighting and derived classes in "/nCoreRtlsNet/Locater"*

2.6.8.3 Configuration parameters of the RTLS functional block

The configuration parameters of the *RTLS* functional block can be consulted remotely for each node and, in some cases, modify their value. These parameters are the next ones:

IMPORTANT NOTE: Coord X, Coord Y and Coord Z parameters are not anymore in the RTLS functional block, as they have been moved to the Kernel block since n-Core® 2.0.

IMPORTANT NOTE: Broadcast Period parameter is not anymore in the RTLS functional block, as it has been moved to the Management block since n-Core® 2.0.

1. **Number of broadcast bursts (*Num Broadcast Sent*):** it is the number of bursts of broadcast frames (see *Broadcast Period* parameter in Mgmt functional block and *Broadcast* messages in this functional block) that will be sent periodically by the *tag*, *router* (to be used by dynamic distance estimators) or *mobile router* (not yet available). By default, this parameter is set to 1 (a unique broadcast burst).
2. **Period between broadcast bursts (*Period Between Broadcast Bursts*)**: it is the elapsed time between sending a broadcast burst and sending the next one, if the number of broadcast bursts (*Num Broadcast Sent* parameter) is higher than 1.

   Note: *In the case of the tags, the device will be awake for that period of time between sending one burst and the next one. In this regard, it is strongly recommended to establish values higher than 100 ms.*

   Note: *the Broadcast Period parameter in the Mgmt functional block and these two last parameters (Num Broadcast Sent and Period Between Broadcast Bursts) are better explained with an example. Let's suppose the next values are established: Broadcast Period = 1000, Num Broadcast Sent = 3 and Period Between Broadcast Bursts = 150. The device (e.g., tag) will send 3 broadcasts bursts, expending 450 ms and during the time the device stays awake. Once the device has sent all the broadcast bursts, the device prepares to sleep and it sleeps for 1000 ms (Broadcast Period). When this broadcast period finishes, the device wakes up and sends 3 broadcast bursts again. This loop will be repeated periodically.*

   Note: *a higher number of broadcast bursts imply a more energy consumption, but the possibilities of locating a tag are increased.*

3. **Tags table period (*Tags Table Period*):** it is the period used by all the devices (except for the tags) to send its detection information to its collecting node (or coordinator). The detection information is a

table composed by, amongst other data, those devices whose broadcasts have been received by the device. It is strongly recommended not to establish a value below 1000 ms.

4. **RTLS device type (*Type Of Device*):** this parameter informs about the RTLS device type. The device can be one of the next RTLS types: *Coordinator*, *Router*, *Router mobile*, *Tag* and *Collecting node*.
   - If the device is defined as <u>*Router*</u> in the Management's *App Device Type*, the device can be defined as *Router* or *Router mobile* for the RTLS device type.
   - If the device is defined as <u>*Coordinator, Tag or Collecting node*</u> in the Management's *App Device Type* this parameter will be read-only and the user cannot modify its RTLS device type.

     Note: *see Annex* **¡Error! No se encuentra el origen de la referencia.** *for the specific Node id values according to the type of device.*

5. **Period to send empty tags tables (*Period Empty Tags Table*):** period with which a router sends its tags table when this table is empty of devices. If the value of this parameter is zero then the device will only send its tag table to its collecting node when this table is not empty. This allows reducing the traffic in the wireless network.
6. **If send LQI information or not (*Send LQI Information*):** Boolean parameter representing if a router will send LQI information in *Tags Table* responses and messages or not.

   Note: *this parameter must set to "true" if you are using a distance estimator that utilizes the LQI value to estimate distances.*

   Note: *the RSSI (Received Signal Strength Indication) and the transmission power used by each device that broadcast its identification are always sent by routers in their Tags Table frames.*

- **n-Core++**
  - *ncRtlsParamId type in "/rtls/rtlsparamstypes.h"*
- **n-Core.C**
  - *ncRtlsParamId_t in "rtls/ncrtls.h"*
- **n-Core.NET**
  - *ncRtlsNetParamId in "/nCoreRtlsNet/IncludenCoreRtlsNet.cs"*

In Table 3, the RTLS parameters and their default values are shown.

Table 3. Default values for the configuration parameters of the *RTLS* functional block.

Parameter	Default value	Read-Only
Num Broadcast Sent	1	-
Period Between Broadcast Bursts	0 ms	-
Tags Table Period	0 ms	-
Type Of Device	Router	• Router/Router mobile: - • Coordinator/Tag/Collector: Yes
Period Empty Tags Table	0 ms	-
Send LQI Information	True	-

2.6.8.4 RTLS command request/response frames

In the RTLS functional block there are the next command request/response frames:

1. **Obtaining a RTLS configuration parameter of a node (*GetParam*)**: it allows obtaining the value of a *RTLS* configuration parameter of a node connected to the ZigBee™ network.

- **n-Core++:**
  - *ncRtls::GetParam() method in "/rtls/rtls.h"*
  - *ncRtlsGetParamCommandResponseEvent class*
- **n-Core.C:**
  - *ncRtlsGetParam() and ncRtlsDispatcherGetParam() functions in "/rtls/ncrtls.h"*
  - *ncCommonGetParamCommandResponseEvent_t struct*
- **n-Core.NET:**
  - *ncRtlsNetGetParam() method in "/nCoreRtlsNet/IncludenCoreRtlsNet.cs"*
  - *ncRtlsNetGetParamCommandResponseEvent class*

2. **Setting or modifying a RTLS configuration parameter of a node (*SetParam*)**: this command allows the user to modify the value of a *RTLS* configuration parameter of a node connected to the ZigBee™ network.
   - **n-Core++:**
     - *ncRtls::SetParam() method in "/rtls/rtls.h"*
     - *ncRtlsSetParamCommandResponseEvent class*
   - **n-Core.C:**
     - *ncRtlsSetParam() and ncRtlsDispatcherSetParam() functions in "/rtls/ncrtls.h"*
     - *ncCommonSetParamCommandResponseEvent_t struct*
   - **n-Core.NET:**
     - *ncRtlsNetSetParam() method in "/nCoreRtlsNet/IncludenCoreRtlsNet.cs"*
     - *ncRtlsNetSetParamCommandResponseEvent class*

3. **Obtaining the tags table of a node (*GetTagsTable*):** it allows the user obtaining the detection information (tags table) of a node (router or router mobile) connected to the network. The tags table (detection information) is composed by:
   - If the node (router) whose location information has been requested is mobile or not.
   - Detection information from those nodes (tags, mobile router or other routers, in the case a dynamic distance estimator mechanism is enabled) whose RTLS *Broadcast* message has been received in the node. For every node detected and stored in the tags table, the next information is included (see *Detected Nodes* in the Common functional block).
     - Tag (or mobile router, or router) identification (see *Node Id* in Management functional block)
     - Tag (or mobile router, or router) power transmission (see *Power Transmission* in Management functional block)
     - RSSI
     - LQI (if *Send LQI Information* parameter is enabled for the router polled).

   Note: *the position (coordinates) of the node (router) whose location information has been requested is not anymore included in this response. Locaters automatically gets this information from the NodeAlive (see Kernel functional block) messages coming from routers.*

   - **n-Core++:**
     - *ncRtls::GetTagsTable() method in "/rtls/rtls.h"*
     - *ncRtlsGetTagsTableCommandResponseEvent class*
   - **n-Core.C:**
     - *ncRtlsGetTagsTable() and ncRtlsDispatcherGetTagsTable() functions in "/rtls/ncrtls.h"*
     - *ncCommonGetTagsTableCommandResponseEvent_t struct*
   - **n-Core.NET:**
     - *ncRtlsNetGetTagsTable() method in "/nCoreRtlsNet/IncludenCoreRtlsNet.cs"*
     - *ncRtlsNetGetTagsTableCommandResponseEvent class*

2.6.8.5 RTLS methods/functions for managing the locating engine

In the RTLS functional block there are the next methods/functions for managing the locating engine:

1. **Starting a location process (*StartLocation*)**: Method/function for starting a location process using a proxy or set of proxies (see *Proxies* in Events functional block) and a certain *Locater* definition (see *Distance Estimators, Weightings and Locaters* in this functional block).
   - **n-Core++:**
     - *ncRtls::StartLocation() method in "/rtls/rtls.h"*
   - **n-Core.C:**
     - *ncRtlsStartLocation() and ncRtlsStartLocationWithProxiesList() functions in "/rtls/ncrtls.h"*
   - **n-Core.NET:**
     - *ncRtlsNetStartLocation() method in "/nCoreRtlsNet/IncludenCoreRtlsNet.cs"*

2. **Stopping a location process (*StopLocation*)**: Method/function for stopping a location process started previously using *StartLocation*.
   - **n-Core++:**
     - *ncRtls::StopLocation() method in "/rtls/rtls.h"*
   - **n-Core.C:**
     - *ncRtlsStopLocation() function in "/rtls/ncrtls.h"*
   - **n-Core.NET:**
     - *ncRtlsNetStopLocation() method in "/nCoreRtlsNet/IncludenCoreRtlsNet.cs"*

3. **Stopping all location processes (*StopAllLocations*)**: Method/function for stopping all location processes started previously using *StartLocation*.
   - **n-Core++:**
     - *ncRtls::StopAllLocations() method in "/rtls/rtls.h"*

4. **Registering a location event handler (*RegisterLocationEventHandler*)**: Method/function for registering an event handler (see *Events* and *Event Handlers* in Events functional block) to receive periodic location events coming from a location process started previously using *StartLocation*.
   - **n-Core++:**
     - *ncRtls::RegisterLocationEventHandler() method in "/rtls/rtls.h"*
     - *ncRtlsLocationEvent class in "/rtls/locater/locationevent.h"*
   - **n-Core.C:**
     - *ncRtlsRegisterLocationEventHandler() and ncRtlsRegisterLocationEventDispatcher() functions in "rtls/ncrtls.h"*
     - *ncRtlsLocationEvent_t struct in "/rtls/ncrtls.h"*
   - **n-Core.NET:**
     - *ncRtlsNetRegisterLocationEventHandler() and ncRtlsNetRegisterLocationEventDispatcher() methods in "/nCoreRtlsNet/IncludenCoreRtlsNet.cs"*
     - *ncRtlsNetLocationEvent in "/nCoreRtlsNet/Events"*

5. **Unregistering a location event handler (*UnregisterLocationEventHandler*)**: Method/function for unregistering an event handler (see *Events* and *Event Handlers* in Events functional block) to stop receiving periodic location events coming from a location process started previously using *StartLocation*.
   - **n-Core++:**

        - *ncRtls::UnregisterLocationEventHandler() method in "/rtls/rtls.h"*
        - *ncRtlsLocationEvent class in "/rtls/locater/locationevent.h"*
    - **n-Core.C:**
        - *ncRtlsUnregisterLocationEventHandler() and ncRtlsUnregisterLocationEventDispatcher() functions in "/rtls/ncrtls.h"*
        - *ncRtlsLocationEvent_t struct in "rtls/ncrtls.h"*
    - **n-Core.NET:**
        - *ncRtlsNetUnregisterLocationEventHandler() and ncRtlsNetUnregisterLocationEventDispatcher() methods in "/nCoreRtlsNet/IncludenCoreRtlsNet.cs"*
        - *ncRtlsNetLocationEvent in "/nCoreRtlsNet/Events"*

2.6.8.6 RTLS message frames

The message frames that exist in this block are:

1. **Broadcast message (*Broadcast*):** this message is sent by tags, mobile routers and routers (in the case a dynamic distance estimator mechanism is enabled). This message is sent with only one hop as time-to-live, so the message will not be spread over the network infinitely. This message will be received by those routers inside its coverage radio.

    Note: *the API does not inform directly about these kinds of messages because they are sent straight between nodes connected to the network.*

    - **n-Core++:**
        - *not available from the API*
    - **n-Core.C:**
        - *not available from the API*
    - **n-Core.NET:**
        - *not available from the API*

2. **Tags table message (*TagsTable*):** router send this message periodically to its collecting node, if the tags table period is higher than zero (see *Tags Table Period* in this functional block). This message contains the tags table (location information) of the router. The tags table (location information) is composed by
    - If the node (router) whose location information is being sent is mobile or not.
    - Detection information from those nodes (tags, mobile router or other routers, in the case a dynamic distance estimator mechanism is enabled) whose RTLS *Broadcast* message has been received in the node. For every node detected and stored in the tags table, the next information is included (see *Detected Nodes* in the Common functional block).
        - Tag (or mobile router, or router) identification (see *Node Id* in Management functional block)
        - Tag (or mobile router, or router) power transmission (see *Power Transmission* in Management functional block)
        - RSSI
        - LQI (if *Send LQI Information* parameter is enabled for the router sending the message).

    Note: *the position (coordinates) of the node (router) whose location information has been requested is not anymore included in this message. Locaters automatically gets this information from the NodeAlive (see Kernel functional block) messages coming from routers.*

__Note:__ the API can inform directly about these messages or they can be treated automatically by any one of the location algorithms (see Locaters) included in the API. In the last case, the user selects by code (see StartLocation command) one of the different location algorithms and the API will automatically estimate the position of the tags and it will inform the user application through location events (see RegisterLocationEventHandler in this functional block).

- **n-Core++:**
  - *ncRtls::RegisterGetTagsTableEventHandler() method in "/rtls/rtls.h"*
  - *ncRtls::UnregisterGetTagsTableEventHandler() method in "/rtls/rtls.h"*
  - *ncRtlsGetTagsTableMessageEvent class*
- **n-Core.C:**
  - *ncRtlsRegisterGetTagsTableEventHandler () and ncRtlsRegisterGetTagsTableEventDispatcher() functions in "/rtls/ncrtls.h"*
  - *ncRtlsUnregisterGetTagsTableEventHandler() and ncRtlsUnregisterGetTagsTableEventDispatcher() functions in "/rtls/ncrtls.h"*
  - *ncRtlsGetTagsTableMessageEvent_t struct*
- **n-Core.NET:**
  - *ncRtlsNetRegisterGetTagsTableEventHandler() and ncRtlsNetRegisterGetTagsTableEventDispatcher() methods in "/nCoreRtlsNet/IncludenCoreRtlsNet.cs"*
  - *ncRtlsNetUnregisterGetTagsTableEventHandler() and ncRtlsNetUnregisterGetTagsTableEventDispatcher() methods in "/nCoreRtlsNet/IncludenCoreRtlsNet.cs"*
  - *ncRtlsNetGetTagsTableMessageEvent in "/nCoreRtlsNet/Events"*

2.6.9 Data Functional Block

The Data functional block is composed by several functionalities related to sending and receiving general-purpose binary data between the nodes connected to the network.

- **n-Core++**
  - *"/data/data.h"*
  - *ncore++_data.dll / libncore++_data.so*
- **n-Core.C**
  - *"data/ncdata.h"*
  - *ncore_data.dll / libncore_data.so*
- **n-Core.NET**
  - *"/nCoreDataNet"*

2.6.9.1 Configuration parameters of the Data functional block

The configuration parameters of the *Data* functional block can be consulted remotely for each node and, in some cases, modify their value. These parameters are the next ones:

1. **Operation mode (*Operation Mode*):** pack of configuration parameters that define how the data messages (see *UnrelData* messages in this functional block) received over-the-air by a Sirius device from other remote Sirius device are forwarded from the receiving Sirius device to the computer or other device connected to this receiving device through a COM port using a USB (virtual COM port in this case) or a RS-232 cable (only present in Sirius A devices).
   - *mode:*
     - *normal*: This operation mode will behave as follows:

        - Data packaged into the *UnrelData* messages received over-the-air by the device will be sent to the USART as the normal way.
    - *sent data by esc char*: This operation mode will behave as follows:
        - Data packaged into the *UnrelData* messages received over-the-air by the device will be sent to the USART without n-Core® encapsulation.
        - Received data from the USART will be packaged into an *UnrelData* message when the defined escape character is received.
- *esc char*: depending on the selected operation *mode*:
    - *normal*: unused byte.
    - *sent data by esc char*: this part will be composed by 1 byte representing an escape character (unsigned 8 bits). Defines the escape character that will be used as limit to sent the data received from the USART. When this escape character is received from the USART, all the previous data received (from the USART) and stored into a buffer will be sent over-the-air to its collecting node, packaged into an *UnrelData* message.

- **n-Core++**
    - *ncDataParamId type in "/data/dataparamstypes.h"*
- **n-Core.C**
    - *ncDataParamId_t in "/data/ncdata.h"*
- **n-Core.NET**
    - *ncDataParamId in "/nCoreDataNet/IncludenCoreDataNet.cs"*

In Table 4, the *Data* parameters and their default values are shown.

Table 4. Default values for the configuration parameters of the *Data* functional block.

Parameter	Default value	Read-only
Operation Mode	mode = NORMAL esc char = (unused)	-

2.6.9.2 Data command request/response frames

The command request and command response frames that exist in this block are the next ones:

1. **Obtaining a Data configuration parameter of a node (*GetParam*)**: it allows obtaining the value of a configuration parameter of a node connected to the ZigBee™ network.
    - **n-Core++:**
        - *ncData::GetParam() method in "/data/data.h"*
        - *ncDataGetParamCommandResponseEvent class*
    - **n-Core.C:**
        - *ncDataGetParam() and ncDataDispatcherGetParam() functions in "/data/ncdata.h"*
        - *ncDataGetParamCommandResponseEvent_t struct*
    - **n-Core.NET:**
        - *ncDataNetGetParam() method in "/nCoreDataNet/IncludenCoreDataNet.cs"*
        - *ncDataNetGetParamCommandResponseEvent class*

2. **Setting or modifying a Data configuration parameter of a node (*SetParam*)**: this command allows the user to modify the value of a configuration parameter of a node connected to the ZigBee™ network.
    - **n-Core++:**
        - *ncData::SetParam() method in "/data/data.h"*

        - *ncDataSetParamCommandResponseEvent class*
    - **n-Core.C:**
        - *ncDataSetParam() and ncDataDispatcherSetParam() functions in "/data/ncdata.h"*
        - *ncDataSetParamCommandResponseEvent_t struct*
    - **n-Core.NET:**
        - *ncDataNetSetParam() method in "/nCoreDataNet/IncludenCoreDataNet.cs"*
        - *ncDataNetSetParamCommandResponseEvent class*

2.6.9.3 Data message frames

Currently, this functional block contains only one message frame:

1. **Unreliable data message (*UnrelData*):** it allows sending general-purpose binary data from one node to another one connected to the same ZigBee™ network. The received binary data is forwarded through the serial port (USB or RS-232) of the device in order that the user application in a local or remote computer can use this data through the API of n-Core®. The maximum amount of data that can be sent in one data message is 70 bytes.
    - **n-Core++:**
        - *ncData::SendUnrelDataMessage() method in "/data/data.h"*
        - *ncData::RegisterUnrelDataMessageEventHandler() method in "/data/data.h"*
        - *ncData::UnregisterUnrelDataMessageEventHandler() method in "/data/data.h"*
        - *ncDataUnrelDataMessageEvent class*
    - **n-Core.C:**
        - *ncDataSendUnrelDataMessage() function in "/data/ncdata.h"*
        - *ncDataRegisterUnrelDataMessageEventHandler() and ncDataRegisterUnrelDataMessageEventDispatcher() functions in "/data/ncdata.h"*
        - *ncDataUnregisterUnrelDataMessageEventHandler() and ncDataUnregisterUnrelDataMessageEventDispatcher() function in "/data/ncdata.h"*
        - *ncDataUnrelDataMessageEvent_t struct*
    - **n-Core.NET:**
        - *ncDataNetSendUnrelDataMessage() method in "/nCoreDataNet/IncludenCoreDataNet.cs"*
        - *ncDataNetRegisterUnrelDataMessageEventHandler() and ncDataNetRegisterUnrelDataMessageEventDispatcher() methods in "/nCoreDataNet/IncludenCoreDataNet.cs"*
        - *ncDataNetUnregisterUnrelDataMessageEventHandler() and ncDataNetUnregisterUnrelDataMessageEventDispatcher() methods in "/nCoreDataNet/IncludenCoreDataNet.cs"*
        - *ncDataNetUnrelDataMessageEvent class*

3 Additional information

Disclaimer

Nebusens believes that all information is correct and accurate at the time of issue. Nebusens reserves the right to make changes to this product without prior notice. Please visit the Nebusens website (www.nebusens.com) for the latest available version.

Nebusens does not assume any responsibility for the use of the described product or convey any license under its patent rights.

Nebusens warrants performance of its products to the specifications applicable at the time of sale in accordance with the sale and use conditions of n-Core®. You can check these conditions on the Nebusens website (www.nebusens.com).

Trademarks

n-Core® and related naming and logos are trademarks of Nebusens, S.L. All other product names, trade names, trademarks, logos or service names are the property of their respective owners.

Technical Support

Technical support is provided by Nebusens, S.L. on demand and in accordance to sale and use conditions agreed. You can check these conditions on the Nebusens website (www.nebusens.com).

We provide you with a support forum (support.nebusens.com) for any question related to the n-Core® platform.

A. Special *Node Id* values for n-Core® Sirius devices

For the correct functioning of the distinct n-Core® features, the n-Core® Sirius devices must have a *Node id* (*see Management functional block*) whitin a specific range according to the type of device (*see App Device Type in the Management functional block and Type of Device in the RTLS functional block*).

Table 5. Ranges of *Node Id* values according to the Mgmt's *App Device Type* and RTLS' *Type of Device*.

App Device Type	Min. Node Id (Dec.)	Max. Node Id (Dec.)	Min. Node Id (Hex.)	Max. Node Id (Hex.)
Coordinator	0	0	0x00	0x00
Router / Collecting node	1	47103	0x01	0xB7FF
Router mobile	47104	49151	0xB800	0xBFFF
Tag	49152	65527	0xC000	0xFFF7

If a device has a *Node id* outside its appropriate range according to its type of device, some of the features that will not work properly are:

- ***Management functional block***
  - Devices (*Coordinators, Collecting nodes, Routers* and *Tags*) will not send *Broadcast* messages.
- ***RTLS functional block***
  - Devices (*Coordinators, Collecting nodes, Routers* and *Mobile routers*) will not send *TagsTable* messages.

B. About the *Sleep Period*

In this Annex it is explained deeply how the *Sleep Period* (see Management functional block) works.

The *Sleep Period* is defined as the time (in milliseconds) during which a n-Core® Sirius device will remain asleep, with a much lower energy consumption than when it is awake.

The n-Core® Sirius devices present the next behavior according to the *Sleep Period*:

- ***Tags* are the only devices that can sleep.** The modification of the *Sleep Period* in the rest of devices (*Coordinators, Routers, Collecting nodes*) does not vary their behavior.

- **The *Sleep Period* (if different from zero) regulates the transmission of the next messages**:
  - ***Kernel* functional block**
    - ***Node Alive* messages**: these messages will be sent only when the device wakes up. Therefore:
      - If the *Node Alive Period* (*see Management functional block*) is less than or equal to the *Sleep Period*, then the *Node Alive* messages will be sent each time the device wakes up. For example:
        - *Sleep Period* = 15000 ms
        - *Node Alive Period* = 10000 ms
        - The *Node Alive* message will be sent each 15000 ms
      - If the *Node Alive Period* is greater than the *Sleep Period*, then the *Node Alive* messages will be sent each time the device wakes up taking to account the relationship between both periods. For example:
        - *Sleep Period* = 15000 ms
        - *Node Alive Period* = 25000 ms
        - The *Node Alive* message will be sent each 30000 ms
  - ***Management* functional block**
    - ***Broadcast* messages:** these messages will be sent only when the device wakes up. Therefore:
      - If the *Broadcast Period* (*see Management functional block*) is less than or equal to the *Sleep Period*, then the *Broadcast* messages will be sent each time the device wakes up. For example:
        - *Sleep Period* = 2000 ms
        - *Broadcast Period* = 1000 ms
        - The *Broadcast* message will be sent each 2000 ms
      - If the *Broadcast Period* is greater than the *Sleep Period*, then the *Broadcast* messages will be sent each time the device wakes up taking to account the relationship between both periods. For example:
        - *Sleep Period* = 1000 ms
        - *Broadcast Period* = 3500 ms
        - The *Broadcast* message will be sent each 4000 ms
  - ***Autom* functional block**
    - ***Define Behavior/Rule* messages:** the periodic behavior/rule messages will be sent only when the device wakes up, **with the exception of those behaviors and rules related to low-level interrupts (LOW_LEVEL IRQs)**. Therefore:
      - If the *period* of the *Behavior/Rule* (*see Autom functional block*) is less than or equal to the *Sleep Period*, then the both the actions related to the

Behavior/Rule and the consequent *Define Behavior/Rule* messages will be performed and sent each time the device wakes up. For example:

  - *Sleep Period* = 10000 ms
  - *Behavior's period* = 2000 ms
  - The *Define Behavior* message will be sent each 10000 ms

- If the *period* of the *Behavior/Rule* (*see Autom functional block*) is greater than the *Sleep Period*, then the both the actions related to the *Behavior/Rule* and the consequent *Define Behavior/Rule* messages will be performed and sent each time the device wakes up taking into account the relationship between both periods. For example:
  - *Sleep Period* = 10000 ms
  - *Behavior's period* = 11000 ms
  - The *Define Behavior* message will be sent each 20000 ms
- In the case of **LOW_LEVEL IRQs**, if such interrupts are triggered outside the defined *Guard time* (*see Autom functional block*), **then in that moment the device is woken up** and the message associated will be sent. If the IRQ is associated to a *Rule*, then the actions belonging to that *Rule* will be performed.

C. Ports in the n-Core® Sirius devices

This Annex details the addresses that must be utilized in the definition of *Behaviors* and *Rules* in the Autom functional block according to the pinout of the different n-Core® Sirius devices.

Table 6. Available ports in n-Core® Sirius A devices.

Port	Port type	Port address (Hex.)
GPO_1	GPO	0x01
GPO_2	GPO	0x02
GPO_3	GPO	0x03
GPO_4	GPO	0x04
GPI_1	GPI	0x01
GPI_2	GPI	0x02
GPI_3	GPI	0x03
GPI_4	GPI	0x04
ADC_1	ADC	0x01
ADC_2	ADC	0x02
IRQ_6 (connected to GPI_3)	IRQ	0x06
IRQ_7 (connected to GPI_4)	IRQ	0x07
RELAY_1	GPO	0x1D
RELAY_2	GPO	0x08
I2C	I2C	-
I2C_CLK	GPI/GPO	0x11
I2C_DATA	GPI/GPO	0x12
USART1_TXD	GPI/GPO	0x13
USART1_RXD	GPI/GPO	0x14
USART1_EXTCLK	GPI/GPO	0x15
USART_RTS	GPI/GPO	0x16
USART_CTS	GPI/GPO	0x17
USART_DTR	GPI/GPO	0x18
ADC_1	GPI/GPO	0x19
ADC_2	GPI/GPO	0x1A
ADC_3	GPI/GPO	0x1B
BAT	GPI/GPO	0x1C
1WR	GPI/GPO	0x1D
USART0_TXD	GPI/GPO	0x1E
USART0_RXD	GPI/GPO	0x1F
USART0_EXTCLK	GPI/GPO	0x20
IRQ_6 (connected to GPI_3)	GPI	0x03
IRQ_7 (connected to GPI_4)	GPI	0x04

Table 7. Available ports in n-Core® Sirius B and D devices.

Port	Port type	Port address (Hex.)
Left button (SW3) – IRQ_6	IRQ	0x06
Left button (SW3)	GPI	0x21
Right button (SW4) – IRQ_7	IRQ	0x07
Right button (SW4)	GPI	0x22
I2C	I2C	-
GPIO_2	GPI/GPO	0x02
GPIO_3	GPI/GPO	0x03
GPIO_4	GPI/GPO	0x04
GPIO_5	GPI/GPO	0x05
GPIO_6	GPI/GPO	0x06
I2C_CLK	GPI/GPO	0x11
I2C_DATA	GPI/GPO	0x12
USART1_TXD	GPI/GPO	0x13
USART_CTS	GPI/GPO	0x17
USART_DTR	GPI/GPO	0x18
ADC_1	GPI/GPO	0x19
ADC_2	GPI/GPO	0x1A
BAT	GPI/GPO	0x1C
1WR	GPI/GPO	0x1D
USART0_TXD	GPI/GPO	0x1E
USART0_RXD	GPI/GPO	0x1F
USART0_EXTCLK	GPI/GPO	0x20

Table 8. Available ports in n-Core® Sirius Quantum and RadIOn devices.

Port	Port type	Port address (Hex.)
Left button (SW1) – IRQ_3	IRQ	0x03
Left button (SW1)	GPI	0x24
Right button (SW2) – IRQ_2	IRQ	0x02
Right button (SW2)	GPI	0x23
Accelerometer Interrupt 1 – IRQ_4	IRQ	0x04
Accelerometer Interrupt 1 – GPI	GPI	0x25
Accelerometer Interrupt 2 – IRQ_5	IRQ	0x05
Accelerometer Interrupt 2 - GPI	GPI	0x26
I2C	I2C	-
I2C_CLK	GPI/GPO	0x11
I2C_DATA	GPI/GPO	0x12
USART1_TXD	GPI/GPO	0x13
USART1_RXD	GPI/GPO	0x14
USART1_EXTCLK	GPI/GPO	0x15
USART_RTS	GPI/GPO	0x16
USART_CTS	GPI/GPO	0x17
ADC_1	GPI/GPO	0x19
ADC_2	GPI/GPO	0x1A
USART0_TXD	GPI/GPO	0x1E
USART0_RXD	GPI/GPO	0x1F
USART0_EXTCLK	GPI/GPO	0x20
SPI_CS	GPI/GPO	0x27
HW_SPI_CS	GPI/GPO	0x28
SPI_SCK	GPI/GPO	0x29
SPI_MOSI	GPI/GPO	0x30
SPI_MISO	GPI/GPO	0x31

www.n-core.info