Altair WinProp 2025.1

Scripting and API Reference Guide

Updated: 05/22/2025

Intellectual Property Rights Notice

This Intellectual Property Rights Notice is exemplary, and therefore not exhaustive, of the intellectual property rights held by Altair Engineering Inc. or its affiliates. Software, other products, and materials of Altair Engineering Inc. or its affiliates are protected under laws of the United States and laws of other jurisdictions.

In addition to intellectual property rights indicated herein, such software, other products, and materials of Altair Engineering Inc. or its affiliates may be further protected by patents, additional copyrights, additional trademarks, trade secrets, and additional other intellectual property rights. For avoidance of doubt, copyright notice does not imply publication. Copyrights in the below are held by Altair Engineering Inc. or its affiliates. Additionally, all non-Altair marks are the property of their respective owners. If you have any questions regarding trademarks or registrations, please contact marketing and legal.

This Intellectual Property Rights Notice does not give you any right to any product, such as software, or underlying intellectual property rights of Altair Engineering Inc. or its affiliates. Usage, for example, of software of Altair Engineering Inc. or its affiliates is governed by and dependent on a valid license agreement.

Altair® HyperWorks®, a Design & Simulation Platform

Altair® AcuSolve® ©1997-2025

Altair® Activate® ©1989-2025

Altair® Automated Reporting Director™ ©2008-2022

Altair® Battery Damage Identifier™ ©2019-2025

Altair® CFD™ ©1990-2025

Altair Compose® ©2007-2025

Altair® ConnectMe™ ©2014-2025

Altair® DesignAI™ ©2022-2025

Altair® DSim® ©2024-2025

Altair® DSim® Cloud ©2024-2025

Altair® DSim® Cloud CLI ©2024-2025

Altair® DSim® Studio ©2024-2025

Altair® EDEM™ ©2005-2025

Altair® EEvision™ ©2018-2025

Altair® ElectroFlo™ ©1992-2025

Altair Embed® ©1989-2025

Altair Embed® SE ©1989-2025

Altair Embed®/Digital Power Designer ©2012-2025

Altair Embed®/eDrives ©2012-2025

Altair Embed® Viewer ©1996-2025

Altair® e-Motor Director™ ©2019-2025

Altair® ESAComp® ©1992-2025

Altair® expertAI™ ©2020-2025

Altair® Feko® ©1999-2025

Altair® FlightStream® ©2017-2025

Altair® Flow Simulator™ ©2016-2025

Altair® Flux® ©1983-2025

Altair® FluxMotor® ©2017-2025

Altair® GateVision PRO™ ©2002-2025

Altair® Geomechanics Director™ ©2011-2022

Altair® HyperCrash® ©2001-2023

Altair® HyperGraph® ©1995-2025

Altair® HyperLife® ©1990-2025

Altair® HyperMesh® ©1990-2025

Altair® HyperMesh® CFD ©1990-2025

Altair® HyperMesh ® NVH ©1990-2025

Altair® HyperSpice™ ©2017-2025

Altair® HyperStudy® ©1999-2025

Altair® HyperView® ©1999-2025

Altair® HyperView Player® ©2022-2025

Altair® HyperWorks® ©1990-2025

Altair® HyperWorks® Design Explorer ©1990-2025

Altair® HyperXtrude® ©1999-2025

Altair® Impact Simulation Director™ ©2010-2022

Altair® Inspire™ ©2009-2025

Altair® Inspire™ Cast ©2011-2025

Altair® Inspire™ Extrude Metal ©1996-2025

Altair® Inspire™ Extrude Polymer ©1996-2025

Altair® Inspire™ Form ©1998-2025

Altair® Inspire™ Mold ©2009-2025

Altair® Inspire™ PolyFoam ©2009-2025

Altair® Inspire™ Print3D ©2021-2025

Altair® Inspire™ Render ©1993-2025

Altair® Inspire™ Studio ©1993-2025

Altair® Material Data Center™ ©2019-2025

Altair® Material Modeler™ ©2019-2025

Altair® Model Mesher Director™ ©2010-2025

Altair® MotionSolve® ©2002-2025

Altair® MotionView® ©1993-2025

Altair® Multi-Disciplinary Optimization Director™ ©2012-2025

Altair® Multiscale Designer® ©2011-2025

Altair® newFASANT™ ©2010-2020

Altair® nanoFluidX® ©2013-2025

Altair® NLView™ ©2018-2025

Altair® NVH Director™ ©2010-2025

Altair® NVH Full Vehicle™ ©2022-2025

Altair® NVH Standard™ ©2022-2025

Altair® OmniV™ ©2015-2025

Altair® OptiStruct® ©1996-2025

Altair® PhysicsAI™ ©2021-2025

Altair® PollEx™ ©2003-2025

Altair® PollEx™ for ECAD ©2003-2025

Altair® PSIM™ ©1994-2025

Altair® Pulse™ ©2020-2025

Altair® Radioss® ©1986-2025

Altair® romAI™ ©2022-2025

Altair® RTLvision PRO™ ©2002-2025

Altair® S-CALC™ ©1995-2025

Altair® S-CONCRETE™ ©1995-2025

Altair® S-FRAME® ©1995-2025

Altair® S-FOUNDATION™ ©1995-2025

Altair® S-LINE™ ©1995-2025

Altair® S-PAD™ ©1995-2025

Altair® S-STEEL™ ©1995-2025

Altair® S-TIMBER™ ©1995-2025

Altair® S-VIEW™ ©1995-2025

Altair® SEAM® ©1985-2025

Altair® shapeAI™ ©2021-2025

Altair® signalAI™ ©2020-2025

Altair® Silicon Debug Tools™ ©2018-2025

Altair® SimLab® ©2004-2025

Altair® SimLab® ST ©2019-2025

Altair® SimSolid® ©2015-2025

Altair® SpiceVision PRO™ ©2002-2025

Altair® Squeak and Rattle Director™ ©2012-2025

Altair® StarVision PRO™ ©2002-2025

Altair® Structural Office™ ©2022-2025

Altair® Sulis™ ©2018-2025

Altair®Twin Activate® ©1989-2025

Altair® UDE™ ©2015-2025

Altair® ultraFluidX® ©2010-2025

Altair® Virtual Gauge Director™ ©2012-2025

Altair® Virtual Wind Tunnel™ ©2012-2025

Altair® Weight Analytics™ ©2013-2022

Altair® Weld Certification Director™ ©2014-2025

Altair® WinProp™ ©2000-2025

Altair® WRAP™ ©1998-2025

Altair® HPCWorks®, a HPC & Cloud Platform

Altair® Allocator™ ©1995-2025

Altair® Access™ ©2008-2025

Altair® Accelerator™ ©1995-2025

Altair® Accelerator™ Plus ©1995-2025

Altair® Breeze™ ©2022-2025

Altair® Cassini™ ©2015-2025

Altair® Control™ ©2008-2025

Altair® Desktop Software Usage Analytics™ (DSUA) ©2022-2025

Altair® FlowTracer™ ©1995-2025

Altair® Grid Engine® ©2001, 2011-2025

Altair® InsightPro™ ©2023-2025

Altair® InsightPro™ for License Analytics ©2023-2025

Altair® Hero™ ©1995-2025

Altair ® Liquid Scheduling™ ©2023-2025

Altair® Mistral™ ©2022-2025

Altair® Monitor™ ©1995-2025

Altair® NavOps® ©2022-2025

Altair® PBS Professional® ©1994-2025

Altair® PBS Works™ ©2022-2025

Altair® Simulation Cloud Suite (SCS) ©2024-2025

Altair® Software Asset Optimization (SAO) ©2007-2025

Altair® Unlimited™ ©2022-2025

Altair® Unlimited Data Analytics Appliance™ ©2022-2025

Altair® Unlimited Virtual Appliance™ ©2022-2025

Altair® RapidMiner®, a Data Analytics & AI Platform

Altair® AI Hub ©2023-2025

Altair® AI Edge™ ©2023-2025

Altair® AI Cloud ©2022-2025

Altair® AI Studio ©2023-2025

Altair® Analytics Workbench™ ©2002-2025

Altair® Graph Lakehouse™ ©2013-2025

Altair® Graph Studio™ ©2007-2025

Altair® Knowledge Hub™ ©2017-2025

Altair® Knowledge Studio® ©1994-2025

Altair® Knowledge Studio® for Apache Spark ©1994-2025

Altair® Knowledge Seeker™ ©1994-2025

Altair® IoT Studio™ ©2002-2025

Altair® Monarch® ©1996-2025

Altair® Monarch® Classic ©1996-2025

Altair® Monarch® Complete™ ©1996-2025

Altair® Monarch® Data Prep Studio ©2015-2025

Altair® Monarch Server™ ©1996-2025

March 5, 2025

Technical Support

Altair's support resources include engaging learning materials, vibrant community forums, intuitive online help resources, how-to guides, and a user-friendly support portal.

Visit Customer Support to learn more about our support offerings.

Contents

Scripts and Application Programming Interface (API)

1

The WinProp application programming interface (API) enables flexible integration into other software tools or post-processing tools.

This chapter covers the following:

1.1 Introduction to Scripting and the WinProp API

The programming language that is integrated with WinProp is C/C++. The application programming interface (API), allows you to control WinProp from an external program.

It is recommended that you familiarise yourself with basic scripting before starting with a new scripting project.

Related concepts
Application Programming Interface (API)

1.2 WinProp API Examples

A number of small examples is available that illustrates how to use the WinProp API.

Note: The examples are located at `%FEKO_HOME%\api\winprop\examples\`[1].

Each WinProp API example consists of the following files:

- *`CMakeLists.txt`*
  The `CMakeLists.txt` file is used to set up a Microsoft Visual Studio solution with all dependencies (linked libraries).
- *`.h`*
  The `.h` is the header file. It is a text file that contains the class and function definitions.
- *`.cpp`*
  The `.cpp` file contains the source code. It is a text file that contains the implementation of the class and function definitions.

1.2.1 Example Prerequisites

Before starting with the WinProp API examples, confirm that the minimum requirements are met.

1. Confirm that you have an Altair Feko 2025.1 installation.
2. Install Microsoft Visual Studio 2015 or later.
3. Download CMake 3.13[2] or later.
4. Be familiar with the C/C++ programming languages.

1.2.2 Running an Example on Microsoft Windows

To run a WinProp API example using Microsoft Windows is a four-step process.

- Use CMake to generate the Microsoft Visual Studio project using `CMakeLists.txt` to set up all the dependencies (linked files).
- Use Microsoft Visual Studio to compile the project into an executable.
- Set the `Path` environment variables to point to the required linked libraries at `%FEKO_HOME%\bin`.
- Run the executable and view the results.

1. where `%FEKO_HOME%` refers to the location of the Altair Feko installation, for example `C:\Program Files\Altair\2025.1\feko`.
2. https://cmake.org/download/

Note: As an example, the workflow will be illustrated using the *Indoor_propagation* example located at:

`%FEKO_HOME%\api\winprop\examples\Indoor_propagation`.

Using CMake to Generate the Microsoft Visual Studio Project

CMake is an open-source, cross-platform tool that allows you to build and package software. It requires `CMakelists.txt` to generate build files.

1. Download CMake from https://cmake.org/download/.
2. Run `cmake-gui.exe`.
   The **CMake** dialog is displayed.

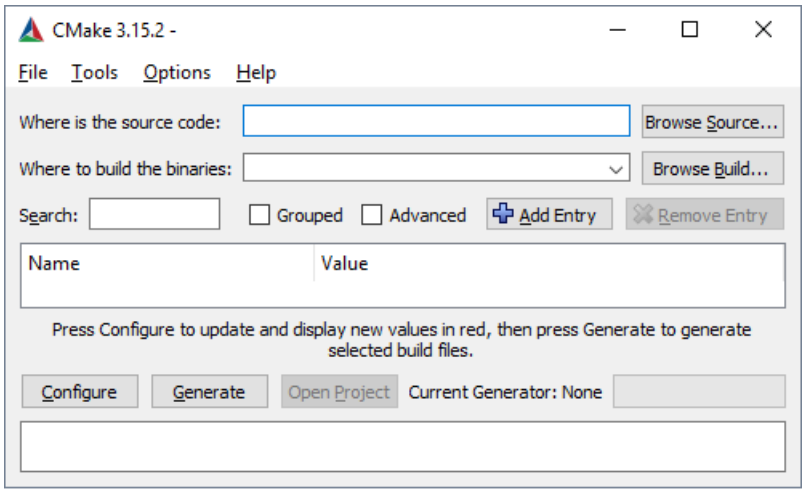

*Figure 1: The **CMake** dialog.*

3. In the **Where is the source code** field, browse to the folder containing `CMakeLists.txt`.

   Note: For example:

   `%FEKO_HOME%\api\winprop\examples\Indoor_propagation`.

4. In the **Where to build the binaries** field, browse to the folder where you want the executable to be built (you might need to create a new folder).

   Note: For example:

   `%FEKO_HOME%\api\winprop\examples\Indoor_propagation\Build`.

5. Click **Configure**.
6. Under **Specify the generator for this project**, select your Microsoft Visual Studio version.
7. Click **Finish**.
8. After the configuration is complete, click **Generate**.

9. Click **Open project** to open the solution in Microsoft Visual Studio.
   Microsoft Visual Studio is launched.
10. Exit **CMake**.

Using the Microsoft Visual Studio Project to Compile the Executable

After CMake was used to generate the Microsoft Visual Studio project with all dependencies (linked libraries), build the project to create an executable.

The following steps assume that you have Microsoft Visual Studio open.

1. [Optional] View the `.cpp` file in the editor by clicking **Solution Explorer** tab (far right of the window) to open the **Solution Explorer** panel and click `indoor_propagation.cpp`.

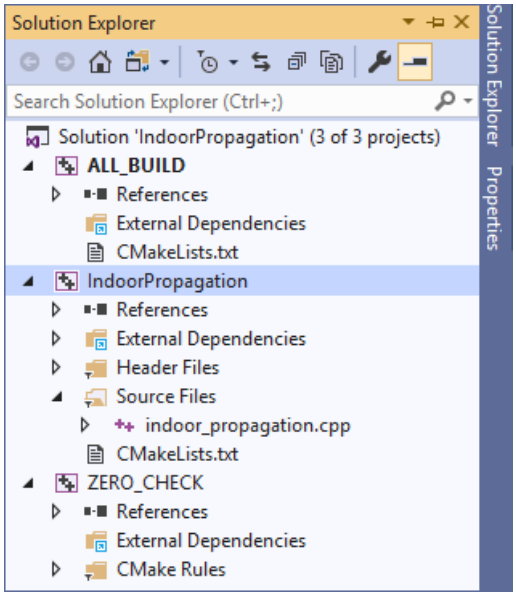

*Figure 2: The **Solution Explorer** panel in Microsoft Visual Studio.*

2. Specify the output folder for the executable.
   a) In the **Solution Explorer** panel, from the **IndoorPropagation** right-click context menu, click **Properties**.
   b) Under **General Properties**, in the **Output Directory** folder, specify the folder where the executable is to be located.

> **Attention:** To avoid the need for administrative rights, select a folder other than `C:\Program Files`.

3. Click **Build** > **Build solution**.
   The result is `IndoorPropagation_debug.exe` in the specified output folder.

Setting the Path Environment Variable

The executable for the WinProp API example requires libraries located in the `%FEKO_HOME%\bin` folder. Set the `Path` environment variable to point to these libraries.

1. In Microsoft Windows, view the environment variables.
2. Edit the `Path` environment variable and add `%FEKO_HOME%\bin`.

Running the Executable and Viewing the Results

Run the example executable and view the simulation results.

1. Run the executable in Microsoft Visual Studio.
   a) On the menu, click **Open** > **Project/Solution** and browse to `IndoorPropagation_debug.exe` and click **Open**.
   b) On the menu, click **Debug** > **Start Without Debugging**[3].

   A console opens and shows the output of the example.
2. Confirm in the console output that all computations are done.

View the simulation results at `%FEKO_HOME%\api\winprop\data\output`.

1.2.3 Running an Example on Linux

Run a WinProp API example using Linux.

The minimum requirements for running an example are:

- Compiler: GCC version 4.9.2 20150212 or equivalent
- CMake version 3.13.1

> **Note:** As an example, the workflow will be illustrated using the *Indoor_propagation* example located at:
>
> `FEKO_HOME/api/winprop/examples/indoor_propagation`.

1. Navigate to the `FEKO_HOME/api/winprop/examples/indoor_propagation`[4] directory using:

   ```
   cd FEKO_HOME/api/winprop/examples/indoor_propagation
   ```

   The `indoor_propagation` directory contains the following three files:

   - `CMakeLists.txt`
   - `indoor_propagation.cpp`
   - `indoor_propagation.h`

3. In the event of a license error, confirm that the environment variable, ALTAIR_LICENSE_PATH, is set. See the Feko Installation Guide for more information.
4. where `%FEKO_HOME%` refers to the location of the Altair Feko installation, for example, `/home/username/2025.1/altair/feko`.

2. Create a new directory called `build` and go into this directory by executing:

```
mkdir build
```

```
cd build
```

3. Set up the project and create a make file by executing:

```
cmake ..
```

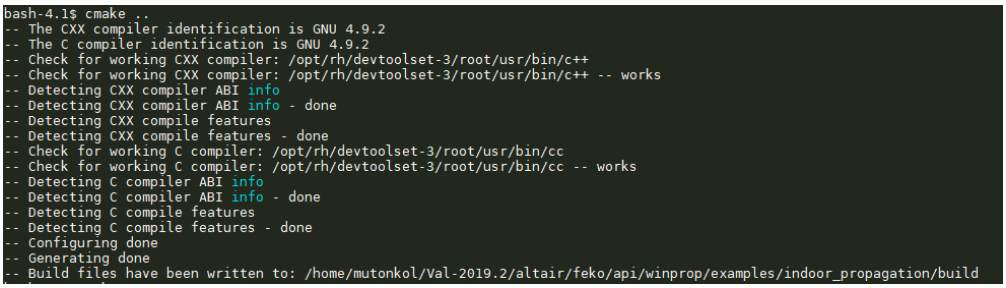

Figure 3: A sample of the output after executing `cmake`.

4. Compile the code in `indoor_propagation.cpp` by executing:

```
make
```

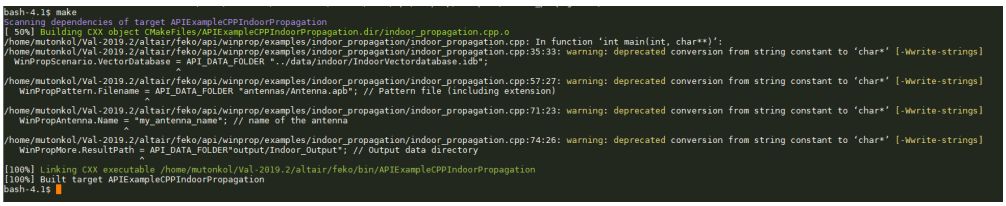

Figure 4: A sample of the output after the code was compiled.

The binary `APIExampleCPPIndoorPropagation` is written to `FEKO_HOME/api/winprop/bin/` directory.

Application Programming Interface (API)

2

The WinProp API enables you to run WinProp simulations from within other software.

This chapter covers the following:

2.1 WinProp Database Conversion Interface

Database conversion interface

This chapter contains the following:

1. Functions
2. Defines and constants
3. Callback functions

2.1.1 Functions

Function List

WinProp_ConvertToBinary
: Converts an indoor ASCII *.ida database to the WinProp binary format.

WinProp_ConvertResultToASCII
: Converts a binary result file to the WinProp ASCII format

WinProp_SetACCallback
: Set callback for AutoCAD converter. The set callback function is called to decide if found 2D objects in a AutoCAD layer shall be included in the conversion or not. It is also possible to set the height of the object and the height above the ground.

WinProp_SetAutoCADLayerSelectionCallback
: Set callback for AutoCAD (*.dwg, *.dxf) to specify which layer to convert.

WinProp_SetHeightLayerSelectionCallback
: Set callback for urban database converters. The set callback function is called for formats, which contain generic properties to define the height of buildings. The callback decides which property contains the building height and allows creation of material definitions for a selected property.

WinProp_Structure_Init_Converter
: This function initialises WinProp_Converter struct with reasonable values

WinProp_Convert
: Converts a given database in a specified format to a database in the appropriate binary WinProp format. The indoor formats are converted to a *.idb, the outdoor formats to a *.odb, topography formats to *.tdb and morpho / clutter formats to a *.mdb database.

WinProp_ConvertMapData
: Conversion of raster topography, clutter data and vector buildings/polygons into the WinProp vector file format (*.idb). Passing vector data to the conversion function is optional but topography and clutter data is required.

Function Details

int ***WinProp_ConvertToBinary****(const char * FilenameIn, const char * FilenameOut)*

Description

Converts an indoor ASCII *.ida database to the WinProp binary format.

Parameters

*const char * FilenameIn*
the filename of the ASCII database in *.ida format to convert.

*const char * FilenameOut*
the filename of the converted binary database.

Returns An integer: 0 = success, otherwise an error.

int ***WinProp_ConvertResultToASCII****(const char * FilenameIn, const char * FilenameOut)*

Description

Converts a binary result file to the WinProp ASCII format

Parameters

*const char * FilenameIn*
the filename of the binary result file to convert.

*const char * FilenameOut*
the filename of the resulting converted ASCII result file.

Returns An integer: 0 = success, otherwise an error.

int ***WinProp_SetACCallback****(*CALLBACK_AC_CONVERTER * *callback)*

Description

Set callback for AutoCAD converter. The set callback function is called to decide if found 2D objects in a AutoCAD layer shall be included in the conversion or not. It is also possible to set the height of the object and the height above the ground.

Parameters

CALLBACK_AC_CONVERTER * *callback*
The callback function.

Returns An integer: 0 = data valid, data not valid otherwise.

int ***WinProp_SetAutoCADLayerSelectionCallback****(*CALLBACK_AUTOCAD_LAYER_SELECTION * *callback)*

Description

Set callback for AutoCAD (*.dwg, *.dxf) to specify which layer to convert.

Parameters

CALLBACK_AUTOCAD_LAYER_SELECTION * *callback*
The callback function.

Returns An integer: 0 = data valid, data not valid otherwise.

int ***WinProp_SetHeightLayerSelectionCallback****(*CALLBACK_HEIGHT_LAYER_SELECTION * *callback)*

Description

Set callback for urban database converters. The set callback function is called for formats, which contain generic properties to define the height of buildings. The callback decides which property contains the building height and allows creation of material definitions for a selected property.

Parameters

CALLBACK_HEIGHT_LAYER_SELECTION * *callback*
The callback function.

Returns An integer: 0 = data valid, data not valid otherwise.

void ***WinProp_Structure_Init_Converter***(WinProp_Converter * *Init)*

Description

This function initialises WinProp_Converter struct with reasonable values

Parameters

WinProp_Converter * *Init*
struct to initialise.

Returns None

int ***WinProp_Convert***(WinProp_Converter * *callConverter,* WinProp_Callback * *callback)*

Description

Converts a given database in a specified format to a database in the appropriate binary WinProp format. The indoor formats are converted to a *.idb, the outdoor formats to a *.odb, topography formats to *.tdb and morpho / clutter formats to a *.mdb database.

Parameters

WinProp_Converter * *callConverter*
Non-null, the settings for the conversion.

WinProp_Callback * *callback*
If non-null, callback for percent and text output.

Returns An integer: 0 = success, otherwise an error.

int ***WinProp_ConvertMapData***(TOPOGRAPHY * *InterfaceTopography,* CLUTTER * *InterfaceClutter,* INDOOR_WALLS * *InterfaceWalls,* MATERIALS * *InterfaceMaterials, const char * OutputFileName,* WinProp_Callback * *Callback)*

Description

Conversion of raster topography, clutter data and vector buildings/polygons into the WinProp vector file format (*.idb). Passing vector data to the conversion function is optional but topography and clutter data is required.

Parameters

TOPOGRAPHY * *InterfaceTopography*
Topography in raster format to be converted to WinProp file format.

CLUTTER * *InterfaceClutter*
Clutter map in raster format and clutter classes to be converted to WinProp file format.

INDOOR_WALLS * *InterfaceWalls*
: Vector data to be converted additionally (optional).

MATERIALS * *InterfaceMaterials*
: Material properties for vector data to be converted (only required if vector data available).

*const char * OutputFileName*
: Name of output file (*.idb).

WinProp_Callback * *Callback*
: Callback for percent output and messages.

Returns An integer: 0 = success, otherwise an error.

The documentation was generated from the following file:

- `source/Interface/Convert.h`

2.1.2 Defines and constants

Detailed Description

Defines and constants

Definition of conversion heights.

CONVERTER_OPTION_IGNORE_ALL *1*
: Ignore all heights during conversion.

CONVERTER_OPTION_SET_HEIGHT *2*
: Set building heights during conversion.

The documentation was generated from the following file:

- `source/Interface/EngineConvert.h`

2.1.3 Callback functions

Detailed Description

Other defines

typedef int(_STD_CALL * CALLBACK_HEIGHT_LAYER_SELECTION) (const char *const *const properties, const int nrProperties, int *const heightPropertyIndex, int *const materialPropertyIndex)
: Type of callback for gathering settings of how to convert urban formats, which have the height described by generic properties.

typedef int(_STD_CALL * CALLBACK_AUTOCAD_LAYER_SELECTION) (const char *const *const layerNames, const int nrLayers, int *const layerConvert, int *const arcDiscretization)

Type of callback for gathering settings of how to convert AutoCAD formats (.dwg, .dxf).

typedef int(_STD_CALL * CALLBACK_AC_CONVERTER) (AC_LayerInfo *layerInfo)

Type of callback for gathering settings of how to convert found 2D objects during conversion of AutoCAD files (dwg, dxf)

The documentation was generated from the following file:

- `source/Interface/EngineConvert.h`

2.2 WinProp Wave Propagation Interface

Wave propagation interface

This chapter contains the following:

1. Functions
2. Outdoor Wave Propagation
3. Callbacks for simulation progress
4. Defines and constants
5. Handling Map and Vector Data

2.2.1 Functions

Functions.

This chapter contains the following:

1. Functions for Allocating Structs
2. Project Management Functions
3. Functions for Wave Propagation Computation
4. Functions for Freeing Allocated Structs
5. Functions for Initialisation of Structures
6. Functions for database pre-processing
7. Functions for computing total power

2.2.2 Functions for Allocating Structs

Function List

WinProp_AllocateResult
: This function allocates memory to the WinProp_Result structure.

WinProp_AllocateResultPlane
: This function allocates memory to the WinProp_ResultPlane structure.

WinProp_Structure_Alloc_Trajectory
: This function frees memory allocated to WinProp_Trajectory.

WinProp_Structure_Alloc_Additional_Freq_Loss
: This function frees memory allocated to WinProp_Additional_Freq_Loss.

WinProp_Structure_Alloc_Polygon
: This function frees memory allocated to WinProp_Polygon.

WinProp_Structure_Alloc_Dynamic_Sets
This function frees memory allocated to WinProp_Dynamic_Sets.

Function Details

int ***WinProp_AllocateResult****(WinProp_Result * Result, int Columns, int Lines, int Heights, const double * HeightsArray, const double * TimeStepsArray)*

Description

This function allocates memory to the WinProp_Result structure.

Parameters

WinProp_Result * *Result*
Result structure.

int Columns
Number of columns in result matrix.

int Lines
Number of rows in result matrix.

int Heights
Number of prediction heights.

*const double * HeightsArray*
Array of prediction heights.

*const double * TimeStepsArray*
Array of time steps.

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_AllocateResultPlane****(WinProp_ResultPlane * Result, int Columns, int Lines)*

Description

This function allocates memory to the WinProp_ResultPlane structure.

Parameters

WinProp_ResultPlane * *Result*
Result structure.

int Columns
Number of columns.

int Lines
Number of lines.

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_Structure_Alloc_Trajectory****(WinProp_Trajectory * ToAlloc, int nrPoints)*

Description

This function frees memory allocated to WinProp_Trajectory.

Parameters

WinProp_Trajectory * *ToAlloc*
: struct to allocate.

int nrPoints
: The number of points to allocate.

Returns An integer: 0.

int ***WinProp_Structure_Alloc_Additional_Freq_Loss***(WinProp_Additional_Freq_Loss * *ToAlloc, int nrEle)*

Description

This function frees memory allocated to WinProp_Additional_Freq_Loss.

Parameters

WinProp_Additional_Freq_Loss * *ToAlloc*
: struct to allocate.

int nrEle
: The number of elements to allocate.

Returns An integer: 0.

int ***WinProp_Structure_Alloc_Polygon***(WinProp_Polygon * *ToAlloc, int nrCorner, int nrIDs)*

Description

This function frees memory allocated to WinProp_Polygon.

Parameters

WinProp_Polygon * *ToAlloc*
: struct to allocate.

int nrCorner
: The number of corners to allocate.

int nrIDs
: The number of IDs to allocate.

Returns An integer: 0.

int ***WinProp_Structure_Alloc_Dynamic_Sets***(WinProp_Dynamic_Sets * *ToAlloc, int nrSets)*

Description

This function frees memory allocated to WinProp_Dynamic_Sets.

Parameters

WinProp_Dynamic_Sets * *ToAlloc*
: struct to allocate.

int nrSets
: The number of sets to allocate.

Returns An integer: 0.

The documentation was generated from the following file:

- `source/Interface/Alloc.h`

2.2.3 Project Management Functions

Function List

WinProp_Open
: Creates a new wave propagation project based on a specific scenario.

WinProp_Close
: Closes the specified propagation project. All memory for prediction results and map data is freed.

WinProp_ErrorText
: Returns the error message corresponding to a given error code.

WinProp_FreeResultID
: Release the result of the last calculation. If there was no calculation before, nothing happens.

WinProp_Version
: This function returns the version of the WinProp API.

WinProp_FilterResult
: This functions filters a predicted result with respect to the selected filter type.

WinProp_FilterResultPlanes
: This functions filters predicted results on arbitrary planes with respect to the selected filter type.

WinProp_Convert_Result
: This function converts a result of a single receiver into a linear array of receivers

WinProp_Convert_Result_Free

WinProp_Releasedate
: This function returns the release date of the WinProp API

WinProp_Result_Read
: Reads a WinProp result into the API result structure.

WinProp_ResultPlanes_Read
: Reads a WinProp plane results into the API result structure.

WinProp_ResultPoints_Read
: Reads WinProp point results into the API result structure.

WinProp_ResultTrajectories_Read
: Reads WinProp trajectory results into the API result structure.

WinProp_RayMatrix_Read
: Reads a WinProp ray matrix from a .ray file into the API result structure.

WinProp_PlanesRayMatrix_Read
: Reads a WinProp ray matrices for planes from a .ray file into the API result structure.

WinProp_Result_Write
: This function writes prediction results to a file.

WinProp_Result_WritePoints
: This function writes point prediction results to a file.

WinProp_Result_WriteTrajectories
: This function writes trajectory prediction results to a file.

WinProp_ResultsRays_Read
: Read WinProp results from the given file(s).

WinProp_ConvertWalls
: Conversion of polygons into walls. Converted file will be stored in WinProp file format.

WinProp_ConvertBuildings
: Conversion of urban buildings into WinProp file format (*.odb).

WinProp_ConvertPixelTopo
: Conversion of topography into WinProp file format (*.tdb).

WinProp_ConvertClutter
: Conversion of clutter into WinProp file format (*.mdb).

WinProp_WriteBitmap
: This function generates a bitmap file with WinProp results.

WinProp_WriteBitmapLegend
: Write bitmap file with legend

WinProp_Legend_Allocate
: This function allocates memory to the WinProp_Legend structure.

WinProp_Legend_Free
: This function frees allocated memory from the WinProp_Legend structure

InterfaceLoadTopoASC
: This function loads Topography information from an ASCII database.

InterfaceLoadClutterASC
: This function loads clutter information from an ASCII database

InterfaceLoadTopoMSI
: This function loads Topography information from a .msi database

InterfaceLoadClutterMSI
: This function loads clutter information from a .msi database

WinProp_ProjectIndoorWrite
: Write indoor project file. Note that this is under active development and the interface may change in future versions.

Function Details

int ***WinProp_Open****(int * DataID, const* WinProp_Scenario ** DataScenario, const* WinProp_Callback ** DataCallback)*

Description

Creates a new wave propagation project based on a specific scenario.

Parameters

*int * DataID*
New handle of the wave propagation project.

const WinProp_Scenario * *DataScenario*
Scenario data (see WinProp_Scenario).

const WinProp_Callback * *DataCallback*
Pointers to callback functions (see WinProp_Callback)..

Returns An integer: 0 = success, otherwise an error.

int ***WinProp_Close****(int DataID)*

Description

Closes the specified propagation project. All memory for prediction results and map data is freed.

Parameters

int DataID
Handle of project to be closed.

Returns An integer: 0 = success, otherwise an error.

int ***WinProp_ErrorText****(unsigned int ID, char * Message, int BufferSize)*

Description

Returns the error message corresponding to a given error code.

Parameters

unsigned int ID
Error number.

*char * Message*
String with error message.

int BufferSize
Size of the buffer.

Returns An integer: 0 = success, otherwise an error.

int ***WinProp_FreeResultID****(int DataID)*

Description

Release the result of the last calculation. If there was no calculation before, nothing happens.

Parameters

int DataID
ID of project (handle of the project see WinProp_Open).

Returns An integer: 0 = success, otherwise an error.

int ***WinProp_Version****(unsigned int * majorRelease, unsigned int * majorUpdate, unsigned int * majorBugfix, char * fullVersion, int fullVersionSize)*

Description

This function returns the version of the WinProp API.

Parameters

*unsigned int * majorRelease*
If non-null, gets set to the major release version.

*unsigned int * majorUpdate*
If non-null, gets set to the major update version.

*unsigned int * majorBugfix*
If non-null, gets set to the bugfix update version.

*char * fullVersion*
If non-null, gets set to the full version string, including a build ID.

int fullVersionSize
The size of the fullVersion buffer.

Returns An integer: 0 = success, otherwise an error.

int ***WinProp_FilterResult****(*WinProp_Result * *Result, int Order, int Type)*

Description

This functions filters a predicted result with respect to the selected filter type.

Parameters

WinProp_Result * *Result*
Result to be filtered.

int Order
Order of filter (must be odd).

int Type
Type of filter: Median (0) or Mean (1).

Returns An integer: 0 = success, otherwise an error.

int ***WinProp_FilterResultPlanes****(*WinProp_ResultPlaneList * *Result, int Order, int Type)*

Description

This functions filters predicted results on arbitrary planes with respect to the selected filter type.

Parameters

WinProp_ResultPlaneList * *Result*
Result to be filtered.

int Order
Order of filter (must be odd).

int Type
Type Median (0) or Mean (1).

Returns An integer: 0 = success, otherwise an error.

int ***WinProp_Convert_Result****(const* WinProp_Result * *Result,* WinProp_Result_2 * *Resultlinear)*

Description

This function converts a result of a single receiver into a linear array of receivers

Parameters

const WinProp_Result * *Result*
Original result.

WinProp_Result_2 * *Resultlinear*
Result in linear array .

Returns An integer: 0 = success, otherwise an error.

int ***WinProp_Convert_Result_Free****(*WinProp_Result_2 * *ResultFree)*

Description

Parameters

WinProp_Result_2 * *ResultFree*
array to be freed.

Returns An integer: 0 = success.

int ***WinProp_Releasedate****(unsigned int * Year, unsigned int * Month, unsigned int * Day)*

Description

This function returns the release date of the WinProp API

Parameters

*unsigned int * Year*
If non-null, the year of release.

*unsigned int * Month*
If non-null, the month of release.

*unsigned int * Day*
If non-null, the day of release.

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_Result_Read****(*WinProp_Result * *Result, const char * Filename)*

Description

Reads a WinProp result into the API result structure.

Parameters

WinProp_Result * *Result*
The result.

*const char * Filename*
Filename of the file to be read.

Returns An int.

int ***WinProp_ResultPlanes_Read****(*WinProp_ResultPlaneList * *Result, const char * Filename)*

Description

Reads a WinProp plane results into the API result structure.

Parameters

WinProp_ResultPlaneList * *Result*
TNon-null, the result.

*const char * Filename*
Filename of of the binary WinProp results.

Returns An int.

int ***WinProp_ResultPoints_Read****(*WinProp_ResultPointsList * *Result, const char * FilenameRes, const char * FilenameRays)*

Description

Reads WinProp point results into the API result structure.

Parameters

WinProp_ResultPointsList * *Result*
Non-null, the result.

*const char * FilenameRes*
The filename of the binary WinProp results.

*const char * FilenameRays*
The filename of the binary .ray file.

Returns An int.

int ***WinProp_ResultTrajectories_Read****(*WinProp_ResultTrajectoryList * *Result, const char * FilenameRes, const char * FilenameRays)*

Description

Reads WinProp trajectory results into the API result structure.

Parameters

WinProp_ResultTrajectoryList * *Result*
Non-null, the result.

*const char * FilenameRes*
The filename of the binary WinProp results.

*const char * FilenameRays*
The filename of the binary .ray file.

Returns An int.

int ***WinProp_RayMatrix_Read****(*WinProp_RayMatrix * *Result, const char * Filename)*

Description

Reads a WinProp ray matrix from a .ray file into the API result structure.

Parameters

WinProp_RayMatrix * *Result*
Non-null, the result.

*const char * Filename*
Filename of the file.

Returns An int.

int ***WinProp_PlanesRayMatrix_Read****(*WinProp_RayMatrixList * *Result, const char * Filename)*

Description

Reads a WinProp ray matrices for planes from a .ray file into the API result structure.

Parameters

WinProp_RayMatrixList * *Result*
Non-null, the result.

*const char * Filename*
Filename of the file.

Returns An int.

int ***WinProp_Result_Write****(int DataID, const* WinProp_Result * *Result, const* WinProp_Antenna * *Transmitter, const char * Filename)*

Description

This function writes prediction results to a file.

Parameters

int DataID
Handle of wave propagation project.

const WinProp_Result * *Result*
Computed result.

const WinProp_Antenna * *Transmitter*
Structure with transmitter details.

*const char * Filename*
name of the file to which results are written.

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_Result_WritePoints****(int DataID, const* WinProp_ResultPointsList * *Result, const* WinProp_Antenna * *Transmitter, const char * Filename)*

Description

This function writes point prediction results to a file.

Parameters

int DataID
Handle of wave propagation project.

const WinProp_ResultPointsList * *Result*
Computed result.

const WinProp_Antenna * *Transmitter*
Structure with transmitter details.

const char * *Filename*
name of the file to which results are written.

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_Result_WriteTrajectories****(int DataID, const* WinProp_ResultTrajectoryList * *Result, const* WinProp_Trajectory * *Trajectories, int NrTrajectories, const* WinProp_Antenna * *Transmitter, const char * Filename)*

Description

This function writes trajectory prediction results to a file.

Parameters

int DataID
Handle of wave propagation project.

const WinProp_ResultTrajectoryList * *Result*
Computed result.

const WinProp_Trajectory * *Trajectories*
Structure with trajectories details.

int NrTrajectories
Number of trajectories.

const WinProp_Antenna * *Transmitter*
Structure with transmitter details.

const char * *Filename*
Name of the file to which results are written.

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_ResultsRays_Read****(*WinProp_Result * *Result,* WinProp_RayMatrix * *ResultRays,* WinProp_ResultPointsList * *WinPropResultPoints,* WinProp_ResultTrajectoryList * *WinPropResultsTrajectories,* WinProp_ResultPlaneList * *WinPropResultsPlanes,* WinProp_RayMatrixList * *WinPropResultRaysPlanes, const bool readRays, const char * Filename, const char * FilenameRays,* WinProp_Callback * *callbacks, const double NotComputedValue)*

Description

Read WinProp results from the given file(s).

Parameters

WinProp_Result * *Result*
If non-null, the area result.

WinProp_RayMatrix * *ResultRays*
If non-null, the area result rays.

WinProp_ResultPointsList * *WinPropResultPoints*
If non-null, the point results.

WinProp_ResultTrajectoryList * *WinPropResultsTrajectories*
If non-null, the trajectories results.

WinProp_ResultPlaneList * *WinPropResultsPlanes*
If non-null, the arbitrary planes results.

WinProp_RayMatrixList * *WinPropResultRaysPlanes*
If non-null, the arbitrary planes result rays.

const bool readRays
True to read rays.

*const char * Filename*
Filename of the file.

*const char * FilenameRays*
If non-null, the filename of the ray file, otherwise ray filename will be derived from Filename.

WinProp_Callback * *callbacks*
If non-null, callbacks to record warning / error messages and progress.

const double NotComputedValue
The value, which should be assigned to not computed pixels.

Returns An int.

int ***WinProp_ConvertWalls****(*INDOOR_WALLS * *InterfaceWalls,* MATERIALS * *InterfaceMaterials, const char * OutputFileName,* WinProp_Callback * *Callback)*

Description

Conversion of polygons into walls. Converted file will be stored in WinProp file format.

Parameters

INDOOR_WALLS * *InterfaceWalls*
Walls to be converted to WinProp file format.

MATERIALS * *InterfaceMaterials*
Materials assigned to vector data.

*const char * OutputFileName*
Name of output file.

WinProp_Callback * *Callback*
Callback for percent output and messages.

Returns An integer: 0 = success, otherwise an error.

int ***WinProp_ConvertBuildings****(*URBAN_BUILDINGS * *InterfaceBuildings, const char * OutputFileName,* WinProp_Callback * *Callback)*

Description

Conversion of urban buildings into WinProp file format (*.odb).

Parameters

URBAN_BUILDINGS * *InterfaceBuildings*
Buildings to be converted to WinProp file format.

*const char * OutputFileName*
Name of output file.

WinProp_Callback * *Callback*
Callback for percent output and messages.

Returns An integer: 0 = success, otherwise an error.

int ***WinProp_ConvertPixelTopo****(*TOPOGRAPHY * *InterfaceTopo, const char * OutputFileName,* WinProp_Callback * *Callback)*

Description

Conversion of topography into WinProp file format (*.tdb).

Parameters

TOPOGRAPHY * *InterfaceTopo*
Topography to be converted to WinProp file format

*const char * OutputFileName*
Name of output file.

WinProp_Callback * *Callback*
Callback for percent output and messages.

Returns An integer: 0 = success, otherwise an error.

int ***WinProp_ConvertClutter****(*CLUTTER * *InterfaceClutter, const char * OutputFileName,* WinProp_Callback * *Callback)*

Description

Conversion of clutter into WinProp file format (*.mdb).

Parameters

CLUTTER * *InterfaceClutter*
Clutter to be converted to WinProp file format

*const char * OutputFileName*
Name of output file.

WinProp_Callback * *Callback*
Callback for percent output and messages.

Returns An integer: 0 = success, otherwise an error.

int ***WinProp_WriteBitmap****(const* WinProp_Result * *Result, double MeterPerPixel, const char * FilenameOutput)*

Description

This function generates a bitmap file with WinProp results.

Parameters

const WinProp_Result * *Result*
Result structure (see WinProp_Result).

double MeterPerPixel
Meter per pixel.

*const char * FilenameOutput*
Name of the bitmap file to be written.

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_WriteBitmapLegend****(const* WinProp_Result * *Result, double MeterPerPixel, const char * FilenameOutput, const* WinProp_Legend * *Legend)*

Description

Write bitmap file with legend

Parameters

const WinProp_Result * *Result*
Result structure (see WinProp_Result).

double MeterPerPixel
Meter per pixel.

*const char * FilenameOutput*
Name of the bitmap file to be written.

const WinProp_Legend * *Legend*
Legend structure (see WinProp_Legend).

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_Legend_Allocate****(*WinProp_Legend * *Legend, int NrSamplingPoints)*

Description

This function allocates memory to the WinProp_Legend structure.

Parameters

WinProp_Legend * *Legend*
Legend structure.

int NrSamplingPoints
Number of sampling points.

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_Legend_Free****(*WinProp_Legend * *Legend)*

Description

This function frees allocated memory from the WinProp_Legend structure

Parameters

WinProp_Legend * *Legend*
Object to be freed.

Returns An integer: 0 = success, failure otherwise.

int ***InterfaceLoadTopoASC****(*TOPOGRAPHY * *Topo, const char * FilenameDatabase)*

Description

This function loads Topography information from an ASCII database.

Parameters

TOPOGRAPHY * *Topo*
Result struct with Topo information.

*const char * FilenameDatabase*
Name of the database file.

Returns An integer: 0 = success, failure otherwise.

int ***InterfaceLoadClutterASC****(*CLUTTER * *Clutter, const char * FilenameDatabase, const char * FilenameCluttertable)*

Description

This function loads clutter information from an ASCII database

Parameters

CLUTTER * *Clutter*
Result struct with Clutter information.

*const char * FilenameDatabase*
Name of the database file.

*const char * FilenameCluttertable*
Name of the clutter table file.

Returns An integer: 0 = success, failure otherwise.

int ***InterfaceLoadTopoMSI****(*TOPOGRAPHY * *Topo, char * FilenameDatabase)*

Description

This function loads Topography information from a .msi database

Parameters

TOPOGRAPHY * *Topo*
Result struct with Topo information.

*char * FilenameDatabase*
Name of the database file.

Returns An integer: 0 = success, failure otherwise.

int ***InterfaceLoadClutterMSI****(*CLUTTER * *Clutter, char * FilenameDatabase, char * FilenameCluttertable)*

Description

This function loads clutter information from a .msi database

Parameters

CLUTTER * *Clutter*
Result struct with Clutter information.

char * *FilenameDatabase*
Name of the database file.

char * *FilenameCluttertable*
Name of the clutter table file.

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_ProjectIndoorWrite****(const* WinProp_Filename *projectFilePath, const* WinProp_Scenario *& winPropScenario, const* WinProp_Area ** winPropArea, const* WinPropPointList_t ** winPropPoints, const* WinPropTrajectoryList_t ** winPropTrajs, const* WinPropAntennaList_t *& winPropAntennas, const* WinProp_Additional *& winPropMore, const* WinProp_Propagation_Results *& outputResults, const* Model_COST ** modelParaCOST, const* Model_DPM ** modelParaDPM, const* Model_RAYTRACING ** modelParaRAYTRACING, const* Model_FREESPACE ** modelParaFREESPACE, const* Model_VRT ** modelParaVRT, const* Model_MOTLEYKEENAN ** modelParaMOTLEYKEENAN, const* WinProp_Additional_Internal ** winPropMoreInternal, const* WinProp_SuperposeMS ** msSuperpose, const* WinProp_MS_Para ** msPara, const* WinProp_MS_AdditionalResults ** msAdditionalResult)*

Description

Write indoor project file. Note that this is under active development and the interface may change in future versions.

Parameters

const WinProp_Filename *projectFilePath*
The path to the project file to write.

const WinProp_Scenario *& winPropScenario*
Structure used to define a scenario (indoor in this case).

const WinProp_Area ** winPropArea*
Structure used to define the prediction area. null if unused.

const WinPropPointList_t ** winPropPoints*
Structure used to define the prediction points. null if unused.

const WinPropTrajectoryList_t ** winPropTrajs*
Structure used to define the prediction trajectories. null if unused.

const WinPropAntennaList_t *& winPropAntennas*
A list of transmitters used in the project. These are of type WinProp_Antenna_Area.

const WinProp_Additional *& winPropMore*
Additional settings for the project.

const WinProp_Propagation_Results *& outputResults*
Output results.

const Model_COST ** modelParaCOST*
(Optional) Settings of the COST model. null if unused.

const Model_DPM ** modelParaDPM*
(Optional) Settings of the DPM model. null if unused.

const Model_RAYTRACING * *modelParaRAYTRACING*
(Optional) Settings of the RAYTRACING model. null if unused.

const Model_FREESPACE * *modelParaFREESPACE*
(Optional) Settings of the FREESPACE model. null if unused.

const Model_VRT * *modelParaVRT*
(Optional) Settings of the VRT model. null if unused.

const Model_MOTLEYKEENAN * *modelParaMOTLEYKEENAN*
(Optional) Settings of the MOTLEY-KEENAN model. null if unused.

const WinProp_Additional_Internal * *winPropMoreInternal*
(Optional) Internal additional settings for the project. null if unused.

const WinProp_SuperposeMS * *msSuperpose*
(Optional) Structure used to define the RunMS superposition of propagation results with given receiving and transmitting array elements. null if unused.

const WinProp_MS_Para * *msPara*
(Optional) Structure used to specify the RunMS computation parameters. null if unused.

const WinProp_MS_AdditionalResults * *msAdditionalResult*
(Optional) Structure used to specify additional RunMS output results. null if unused.

Returns An integer: 0 = success, failure otherwise.

The documentation was generated from the following file:

- `source/Interface/Engine.h`

2.2.4 Functions for Wave Propagation Computation

Function List

WinProp_Predict
Start a prediction of wave propagation coverage.

WinProp_Predict_Planes
Compute a wave propagation prediction on arbitrary planes in a scenario generated with WinProp_Open. Optional input parameters should be set to NULL if not used.

WinProp_Predict_Points
Compute wave propagation prediction between the antenna and given points. Scenario has to be opened with WinProp_Open before.

WinProp_Predict_Trajectories
Compute wave propagation prediction for a given trajectory. Scenario has to be opened with WinProp_Open before.

WinProp_Calibrate_DPM
This function is used to calibrate the dominant path model.

WinProp_Result_GetAngles
: This function calculates elevation/azimuth angles at the BS or MS.

WinProp_Coverage_Analysis
: Computes an interpolated map based on measurement data for a specified scenario opened with WinProp_Open.

Function Details

int ***WinProp_Predict****(int DataID, const* WinProp_Antenna * *DataAntenna, const* WinProp_Area * *DataArea, const* WinProp_Receiver * *DataReceiver, int NrReceiver, const void * DataModelSettings, const* WinProp_Additional * *Additional,* WinProp_Result ** *DataCoverageOut,* WinProp_Result ** *DataDelayOut,* WinProp_Result ** *DataNatureOfPathOut,* WinProp_RayMatrix ** *DataRaysOut,* WinProp_ResultPlaneList ** *PlanesResultOut,* WinProp_RayMatrixList ** *PlanesRaysOut, const* WinProp_Measurements * *Measurements)*

Description

Start a prediction of wave propagation coverage.

Parameters

int DataID
: Handle of wave propagation project.

const WinProp_Antenna * *DataAntenna*
: Configuration of antenna.

const WinProp_Area * *DataArea*
: Definition of prediction area.

const WinProp_Receiver * *DataReceiver*
: If Non-null, represents the receiving points at repeaters antennas.

int NrReceiver
: Number of receiving points at repeaters antennas.

*const void * DataModelSettings*
: Model settings (optional) of corresponding wave propagation model (see Model_DPM, Model_COST or Model_RAYTRACING).

const WinProp_Additional * *Additional*
: Definition of additional settings (optional).

WinProp_Result ** *DataCoverageOut*
: Coverage prediction result.

WinProp_Result ** *DataDelayOut*
: Delay prediction result.

WinProp_Result ** *DataNatureOfPathOut*
: Result structure with path information.

WinProp_RayMatrix ** *DataRaysOut*
: Ray matrix.

WinProp_ResultPlaneList ** *PlanesResultOut*
: If non-null, result structure containing the prediction result for the arbitrary prediction planes defined in the database.

WinProp_RayMatrixList ** *PlanesRaysOut*
: If non-null, result structure containing the ray matrix for the arbitrary prediction planes defined in the database.

const WinProp_Measurements * *Measurements*
: Measurement data for calibration (optional).

Returns An integer: 0 = success, otherwise an error.

int ***WinProp_Predict_Planes****(int DataID, const* WinProp_Antenna * *DataAntenna, int NrPlaneAreas, const* WinProp_AreaPlane * *DataAreas, const void * DataModelSettings, const* WinProp_Additional * *Additional,* WinProp_ResultPlaneList ** *DataCoverageOut,* WinProp_ResultPlaneList ** *DataDelayOut,* WinProp_ResultPlaneList ** *DataNatureOfPathOut,* WinProp_RayMatrixList ** *DataRaysOut, const* WinProp_Measurements * *Measurements)*

Description

Compute a wave propagation prediction on arbitrary planes in a scenario generated with WinProp_Open. Optional input parameters should be set to NULL if not used.

Parameters

int DataID
: Handle of wave propagation project.

const WinProp_Antenna * *DataAntenna*
: Configuration of antenna (see WinProp_Antenna).

int NrPlaneAreas
: Number of arbitrary prediction areas (prediction planes).

const WinProp_AreaPlane * *DataAreas*
: Definition of arbitrary prediction areas (prediction planes) (see WinProp_AreaPlane).

*const void * DataModelSettings*
: Model settings (optional) of corresponding wave propagation model (Model_DPM, Model_COST or Model_RAYTRACING).

const WinProp_Additional * *Additional*
: Definition of additional settings (optional) WinProp_Additional.

WinProp_ResultPlaneList ** *DataCoverageOut*
: Coverage prediction result (see WinProp_ResultPlaneList).

WinProp_ResultPlaneList ** *DataDelayOut*
: Delay prediction result (see WinProp_ResultPlaneList).

WinProp_ResultPlaneList ** *DataNatureOfPathOut*
: Path type result (see WinProp_ResultPlaneList).

WinProp_RayMatrixList ** *DataRaysOut*
: Propagation paths (see WinProp_RayMatrixList).

const WinProp_Measurements * *Measurements*
Measurement data for calibration (optional). This is not yet supported for prediction planes.

Returns An integer: 0 = success, otherwise an error.

int ***WinProp_Predict_Points****(int DataID, const* WinProp_Antenna * *DataAntenna, const* WinProp_Receiver * *DataReceiver, int NrReceiver, const void * DataModelSettings, const* WinProp_Additional * *Additional,* WinProp_ResultPointsList ** *DataResultOut,* WinProp_ResultPlaneList ** *PlanesResultOut,* WinProp_RayMatrixList ** *PlanesRaysOut)*

Description

Compute wave propagation prediction between the antenna and given points. Scenario has to be opened with WinProp_Open before.

Parameters

int DataID
Handle of wave propagation project.

const WinProp_Antenna * *DataAntenna*
Non-null, configuration of antenna.

const WinProp_Receiver * *DataReceiver*
Non-null, array of receiving points, of length equal to NrReceiver.

int NrReceiver
Number of receiving points.

*const void * DataModelSettings*
If non-null, model settings (optional) of corresponding wave propagation model. Needs to be set to the to WinProp_Antenna::Model corresponding type: WINPROP_MODEL_INDOOR.

const WinProp_Additional * *Additional*
If non-null, definition of additional settings (optional).

WinProp_ResultPointsList ** *DataResultOut*
Non-null, structure containing the points results.

WinProp_ResultPlaneList ** *PlanesResultOut*
If non-null, structure containing the prediction result for the arbitrary prediction planes defined in the database.

WinProp_RayMatrixList ** *PlanesRaysOut*
If non-null, result structure containing the ray matrix for the arbitrary prediction planes defined in the database.

Returns An integer: 0 = success, otherwise an error.

int ***WinProp_Predict_Trajectories****(int DataID, const* WinProp_Antenna * *DataAntenna, const* WinProp_Trajectory * *DataTrajectories, int NrTrajectories, const void * DataModelSettings, const* WinProp_Additional * *Additional,* WinProp_ResultTrajectoryList ** *DataResultOut,* WinProp_ResultPlaneList ** *PlanesResultOut,* WinProp_RayMatrixList ** *PlanesRaysOut)*

Description

Compute wave propagation prediction for a given trajectory. Scenario has to be opened with WinProp_Open before.

Parameters

int DataID
Handle of wave propagation project.

const WinProp_Antenna * *DataAntenna*
Non-null, configuration of antenna.

const WinProp_Trajectory * *DataTrajectories*
Non-null, the trajectories. Array of size NrTrajectories.

int NrTrajectories
The number of trajectories.

*const void * DataModelSettings*
If non-null, model settings (optional) of corresponding wave propagation model. Needs to be set to the to WinProp_Antenna::Model corresponding type: WINPROP_MODEL_INDOOR.

const WinProp_Additional * *Additional*
If non-null, definition of additional settings (optional).

WinProp_ResultTrajectoryList ** *DataResultOut*
Non-null, structure containing the trajectory results.

WinProp_ResultPlaneList ** *PlanesResultOut*
If non-null, structure containing the prediction result for the arbitrary prediction planes defined in the database.

WinProp_RayMatrixList ** *PlanesRaysOut*
If non-null, result structure containing the ray matrix for the arbitrary prediction planes defined in the database.

Returns An int.

int ***WinProp_Calibrate_DPM****(*WinProp_Calib_DPM * *Calibration)*

Description

This function is used to calibrate the dominant path model.

Parameters

WinProp_Calib_DPM * *Calibration*
Calibration data structure. See WinProp_Calib_DPM.

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_Result_GetAngles****(int DataID, int x, int y, int z, int Ray, double * AzimuthBS, double * ElevationBS, double * AzimuthMS, double * ElevationMS)*

Description

This function calculates elevation/azimuth angles at the BS or MS.

Parameters

int DataID
ID of the project. See the first parameter in WinProp_Open.

int x
x coordinate matrix index.

int y
y coordinate matrix index.

int z
z coordinate matrix index.

int Ray
Ray index.

*double * AzimuthBS*
Azimuth (phi) at BS.

*double * ElevationBS*
Elevation (theta) at BS.

*double * AzimuthMS*
Azimuth (phi) at MS.

*double * ElevationMS*
Elevation (theta) at MS.

Returns An int.

int ***WinProp_Coverage_Analysis****(int In_DataID, const* WinProp_Area ** In_DataArea, const* WinProp_Surveydata ** In_DataSurvey, const* WinProp_Coverage_Para ** In_CoveragePara,* WinProp_Result *** Out_DataResult)*

Description

Computes an interpolated map based on measurement data for a specified scenario opened with WinProp_Open.

Parameters

int In_DataID
Handle of wave propagation project.

const WinProp_Area ** In_DataArea*
Definition of prediction area (see WinProp_Area).

const WinProp_Surveydata ** In_DataSurvey*
Measurement data (see WinProp_Surveydata).

const WinProp_Coverage_Para ** In_CoveragePara*
Configuration of interpolation algorithm (see WinProp_Coverage_Para).

WinProp_Result *** Out_DataResult*
Computed coverage (see WinProp_Result).

Returns An integer: 0 = success, failure otherwise.

The documentation was generated from the following file:

- `source/Interface/Engine.h`

2.2.5 Functions for Freeing Allocated Structs

Function List

WinProp_Structure_Free_Scenario
: This function releases dynamically allocated data of a WinProp_Scenario struct.

WinProp_Structure_Free_Antenna
: This function releases dynamically allocated data of a WinProp_Antenna struct.

WinProp_Structure_Free_Additional
: This function releases dynamically allocated data of a WinProp_Additional struct.

WinProp_Structure_Free_Model_DPM
: This function releases dynamically allocated data of a Model_DPM struct.

WinProp_Structure_Free_Model_VRT
: This function releases dynamically allocated data of a Model_VRT struct.

WinProp_Structure_Free_Area
: This function releases dynamically allocated data of a WinProp_Area struct.

WinProp_Structure_Free_AreaPlane
: This function releases dynamically allocated data of a WinProp_AreaPlane struct.

WinProp_Structure_Free_Pattern
: This function releases dynamically allocated data of a WinProp_Pattern struct.

WinProp_Structure_Free_Calib
: This function releases dynamically allocated data of a WinProp_Calib_DPM struct.

WinProp_Structure_Free_Coverage_Para
: This function releases dynamically allocated data of a WinProp_Coverage_Para struct.

WinProp_Structure_Free_Result
: This function releases dynamically allocated data of a WinProp_Result struct.

WinProp_Structure_Free_ResultPlane
: This function releases dynamically allocated data of a WinProp_ResultPlane struct.

WinProp_Structure_Free_ResultPlaneList
: This function releases dynamically allocated data of a WinProp_ResultPlaneList struct.

WinProp_Structure_Free_Receiver
: This function releases dynamically allocated data of a WinProp_Receiver struct.

WinProp_Structure_Free_ResultPointsList
: This function releases dynamically allocated data of a WinProp_ResultPointsList struct.

WinProp_Structure_Free_ResultPoint
: This function releases dynamically allocated data of a WinProp_ResultPoint struct.

WinProp_Pattern_Free
: This function releases dynamically allocated data of a WinProp_Pattern struct.

WinProp_Structure_Free_ParameterHybrid
: This function releases dynamically allocated data of a WinProp_ParaHybrid struct.

WinProp_Structure_Free_ParameterMain
: This function releases dynamically allocated data of a WinProp_ParaMain struct.

WinProp_Structure_Free_ParameterRural
: This function releases dynamically allocated data of a WinProp_ParaRural struct.

WinProp_Structure_Free_Measurement
: This function releases dynamically allocated data of a WinProp_Measurement struct.

WinProp_Structure_Free_MeasurementPoint
: This function releases dynamically allocated data of a WinProp_MeasurementPoint struct.

WinProp_Structure_Free_Model_UrbanIRT
: This function releases dynamically allocated data of a Model_UrbanIRT struct.

WinProp_Structure_Free_Model_RayTracing
: This function releases dynamically allocated data of a Model_RAYTRACING struct.

WinProp_Structure_Free_PreProUrban
: This function releases dynamically allocated data of a WinProp_PreProUrban struct.

WinProp_Structure_Free_ResultMonteCarlo
: This function releases dynamically allocated data of a WinProp_ResultMonteCarlo struct.

WinProp_Structure_Free_Trajectory
: This function releases dynamically allocated data of a WinProp_Trajectory struct.

WinProp_Structure_Free_ResultTrajectory
: This function releases dynamically allocated data of a WinProp_ResultTrajectory struct.

WinProp_Structure_Free_ResultTrajectoryList
: This function releases dynamically allocated data of a WinProp_ResultTrajectoryList struct.

WinProp_Structure_Free_Additional_Freq_Loss
: This function releases dynamically allocated data of a WinProp_Additional_Freq_Loss struct.

WinProp_Structure_Free_Polygon
: This function releases dynamically allocated data of a WinProp_Polygon struct.

WinProp_Structure_Free_Polygons
: This function releases multiple dynamically allocated WinProp_Polygon structs.

WinProp_Structure_Free_Dynamic_Sets
: This function frees memory allocated to WinProp_Dynamic_Sets.

WinProp_Group_Dynamics_Free
: This function frees memory allocated to WinProp_Group_Dynamics.

WinProp_FreeResult
: This function releases dynamically allocated data of WinProp_Result structure.

WinProp_FreeRayMatrix
: This function releases dynamically allocated data of WinProp_RayMatrix

WinProp_FreeRayMatrixElement
: This function releases dynamically allocated data of WinProp_RayMatrixElement

WinProp_Structure_Free_RayMatrixList
: This function releases dynamically allocated data of a WinProp_RayMatrixList struct.

Function Details

int ***WinProp_Structure_Free_Scenario****(*WinProp_Scenario * *ToFree)*

Description

This function releases dynamically allocated data of a WinProp_Scenario struct.

Parameters

WinProp_Scenario * *ToFree*
: struct to be freed.

Returns An int.

int ***WinProp_Structure_Free_Antenna****(*WinProp_Antenna * *ToFree)*

Description

This function releases dynamically allocated data of a WinProp_Antenna struct.

Parameters

WinProp_Antenna * *ToFree*
: struct to be freed.

Returns An int.

int ***WinProp_Structure_Free_Additional****(*WinProp_Additional * *ToFree)*

Description

This function releases dynamically allocated data of a WinProp_Additional struct.

Parameters

WinProp_Additional * *ToFree*
: struct to be freed.

Returns An int.

int ***WinProp_Structure_Free_Model_DPM****(*Model_DPM * *ToFree)*

Description

This function releases dynamically allocated data of a Model_DPM struct.

Parameters

Model_DPM * *ToFree*
: struct to be freed.

Returns An int.

int ***WinProp_Structure_Free_Model_VRT***(Model_VRT * *ToFree*)

Description

This function releases dynamically allocated data of a Model_VRT struct.

Parameters

Model_VRT * *ToFree*
struct to be freed.

Returns An int.

int ***WinProp_Structure_Free_Area***(WinProp_Area * *ToFree*)

Description

This function releases dynamically allocated data of a WinProp_Area struct.

Parameters

WinProp_Area * *ToFree*
struct to be freed.

Returns An int.

int ***WinProp_Structure_Free_AreaPlane***(WinProp_AreaPlane * *ToFree*)

Description

This function releases dynamically allocated data of a WinProp_AreaPlane struct.

Parameters

WinProp_AreaPlane * *ToFree*
struct to be freed.

Returns An int.

int ***WinProp_Structure_Free_Pattern***(WinProp_Pattern * *ToFree*)

Description

This function releases dynamically allocated data of a WinProp_Pattern struct.

Parameters

WinProp_Pattern * *ToFree*
struct to be freed.

Returns An int.

int ***WinProp_Structure_Free_Calib***(WinProp_Calib_DPM * *ToFree*)

Description

This function releases dynamically allocated data of a WinProp_Calib_DPM struct.

Parameters

WinProp_Calib_DPM * *ToFree*
struct to be freed.

Returns An int.

int ***WinProp_Structure_Free_Coverage_Para***(WinProp_Coverage_Para * *ToFree*)

Description

This function releases dynamically allocated data of a WinProp_Coverage_Para struct.

Parameters

WinProp_Coverage_Para * *ToFree*
struct to be freed.

Returns An int.

int ***WinProp_Structure_Free_Result***(WinProp_Result * *ToFree*)

Description

This function releases dynamically allocated data of a WinProp_Result struct.

Parameters

WinProp_Result * *ToFree*
struct to be freed.

Returns An int.

int ***WinProp_Structure_Free_ResultPlane***(WinProp_ResultPlane * *ToFree*)

Description

This function releases dynamically allocated data of a WinProp_ResultPlane struct.

Parameters

WinProp_ResultPlane * *ToFree*
struct to be freed.

Returns An int.

int ***WinProp_Structure_Free_ResultPlaneList***(WinProp_ResultPlaneList * *ToFree*)

Description

This function releases dynamically allocated data of a WinProp_ResultPlaneList struct.

Parameters

WinProp_ResultPlaneList * *ToFree*
struct to be freed.

Returns An int.

int ***WinProp_Structure_Free_Receiver***(WinProp_Receiver * *ToFree*)

Description

This function releases dynamically allocated data of a WinProp_Receiver struct.

Parameters

WinProp_Receiver * *ToFree*
struct to be freed.

Returns An int.

int ***WinProp_Structure_Free_ResultPointsList****(*WinProp_ResultPointsList * *ToFree)*

Description

This function releases dynamically allocated data of a WinProp_ResultPointsList struct.

Parameters

WinProp_ResultPointsList * *ToFree*
struct to be freed.

Returns An int.

int ***WinProp_Structure_Free_ResultPoint****(*WinProp_ResultPoint * *ToFree)*

Description

This function releases dynamically allocated data of a WinProp_ResultPoint struct.

Parameters

WinProp_ResultPoint * *ToFree*
struct to be freed.

Returns An int.

int ***WinProp_Pattern_Free****(*WinProp_Pattern * *ToFree)*

Description

This function releases dynamically allocated data of a WinProp_Pattern struct.

Parameters

WinProp_Pattern * *ToFree*
struct to be freed.

Returns An int.

int ***WinProp_Structure_Free_ParameterHybrid****(*WinProp_ParaHybrid * *Para)*

Description

This function releases dynamically allocated data of a WinProp_ParaHybrid struct.

Parameters

WinProp_ParaHybrid * *Para*
struct to be freed.

Returns An int.

int ***WinProp_Structure_Free_ParameterMain****(*WinProp_ParaMain * *ParameterUrban)*

Description

This function releases dynamically allocated data of a WinProp_ParaMain struct.

Parameters

WinProp_ParaMain * *ParameterUrban*
struct to be freed.

Returns An int.

int ***WinProp_Structure_Free_ParameterRural***(WinProp_ParaRural * *Para*)

Description

This function releases dynamically allocated data of a WinProp_ParaRural struct.

Parameters

WinProp_ParaRural * *Para*
struct to be freed.

Returns An int.

int ***WinProp_Structure_Free_Measurement***(WinProp_Measurement * *Para*)

Description

This function releases dynamically allocated data of a WinProp_Measurement struct.

Parameters

WinProp_Measurement * *Para*
struct to be freed.

Returns An int.

int ***WinProp_Structure_Free_MeasurementPoint***(WinProp_MeasurementPoint * *Point*)

Description

This function releases dynamically allocated data of a WinProp_MeasurementPoint struct.

Parameters

WinProp_MeasurementPoint * *Point*
struct to be freed.

Returns An int.

int ***WinProp_Structure_Free_Model_UrbanIRT***(Model_UrbanIRT * *ParaIRT*)

Description

This function releases dynamically allocated data of a Model_UrbanIRT struct.

Parameters

Model_UrbanIRT * *ParaIRT*
struct to be freed.

Returns An int.

int ***WinProp_Structure_Free_Model_RayTracing***(Model_RAYTRACING * *ToFree*)

Description

This function releases dynamically allocated data of a Model_RAYTRACING struct.

Parameters

Model_RAYTRACING * *ToFree*
struct to be freed.

Returns An int.

int ***WinProp_Structure_Free_PreProUrban***(WinProp_PreProUrban * *ToFree*)

Description

This function releases dynamically allocated data of a WinProp_PreProUrban struct.

Parameters

WinProp_PreProUrban * *ToFree*
struct to be freed.

Returns An int.

int ***WinProp_Structure_Free_ResultMonteCarlo***(WinProp_ResultMonteCarlo * *ToFree*)

Description

This function releases dynamically allocated data of a WinProp_ResultMonteCarlo struct.

Parameters

WinProp_ResultMonteCarlo * *ToFree*
struct to be freed.

Returns An int.

int ***WinProp_Structure_Free_Trajectory***(WinProp_Trajectory * *ToFree*)

Description

This function releases dynamically allocated data of a WinProp_Trajectory struct.

Parameters

WinProp_Trajectory * *ToFree*
struct to be freed.

Returns An int.

int ***WinProp_Structure_Free_ResultTrajectory***(WinProp_ResultTrajectory * *ToFree*)

Description

This function releases dynamically allocated data of a WinProp_ResultTrajectory struct.

Parameters

WinProp_ResultTrajectory * *ToFree*
struct to be freed.

Returns An int.

int ***WinProp_Structure_Free_ResultTrajectoryList***(WinProp_ResultTrajectoryList * *ToFree*)

Description

This function releases dynamically allocated data of a WinProp_ResultTrajectoryList struct.

Parameters

WinProp_ResultTrajectoryList * *ToFree*
struct to be freed.

Returns An int.

int ***WinProp_Structure_Free_Additional_Freq_Loss***(WinProp_Additional_Freq_Loss * *ToFree)*

Description

This function releases dynamically allocated data of a WinProp_Additional_Freq_Loss struct.

Parameters

WinProp_Additional_Freq_Loss * *ToFree*
struct to be freed.

Returns An int.

int ***WinProp_Structure_Free_Polygon***(WinProp_Polygon * *ToFree)*

Description

This function releases dynamically allocated data of a WinProp_Polygon struct.

Parameters

WinProp_Polygon * *ToFree*
struct to be freed.

Returns An int.

int ***WinProp_Structure_Free_Polygons***(WinProp_Polygon ** *ToFree, int NrPolygons)*

Description

This function releases multiple dynamically allocated WinProp_Polygon structs.

Parameters

WinProp_Polygon ** *ToFree*
struct to be freed.

int NrPolygons
Number of polygons.

Returns An int.

int ***WinProp_Structure_Free_Dynamic_Sets***(WinProp_Dynamic_Sets * *ToFree)*

Description

This function frees memory allocated to WinProp_Dynamic_Sets.

Parameters

WinProp_Dynamic_Sets * *ToFree*
struct to free.

Returns An integer: 0.

int ***WinProp_Group_Dynamics_Free***(WinProp_Group_Dynamics * *ToFree)*

Description

This function frees memory allocated to WinProp_Group_Dynamics.

Parameters

WinProp_Group_Dynamics * *ToFree*
struct to free.

Returns An integer: 0.

int ***WinProp_FreeResult***(WinProp_Result * *ToFree)*

Description

This function releases dynamically allocated data of WinProp_Result structure.

Parameters

WinProp_Result * *ToFree*
struct to be freed.

Returns An int.

int ***WinProp_FreeRayMatrix***(WinProp_RayMatrix * *ToFree)*

Description

This function releases dynamically allocated data of WinProp_RayMatrix

Parameters

WinProp_RayMatrix * *ToFree*
struct x to be freed.

Returns An int.

int ***WinProp_FreeRayMatrixElement***(WinProp_RayMatrixElement * *ToFree)*

Description

This function releases dynamically allocated data of WinProp_RayMatrixElement

Parameters

WinProp_RayMatrixElement * *ToFree*
struct to be freed.

Returns An int.

int ***WinProp_Structure_Free_RayMatrixList***(WinProp_RayMatrixList * *ToFree)*

Description

This function releases dynamically allocated data of a WinProp_RayMatrixList struct.

Parameters

WinProp_RayMatrixList * *ToFree*
struct to be freed.

Returns An int.

The documentation was generated from the following file:

- `source/Interface/Free.h`

2.2.6 Functions for Initialisation of Structures

Function List

WinProp_Structure_Init_Callback
: This function initialises callback functions

WinProp_Structure_Init_Scenario
: This function initialises WinProp_Scenario struct with reasonable values

WinProp_Structure_Init_Dynamic_Antenna
: This function initialises WinProp_Dynamic_Antenna struct with reasonable values

WinProp_Structure_Init_Antenna
: This function initialises WinProp_Antenna struct with reasonable values

WinProp_Structure_Init_Carrier
: This function initialises WinProp_Carrier struct with reasonable values

WinProp_Structure_Init_Propagation_Results
: This function initialises WinProp_Propagation_Results struct with reasonable values

WinProp_Structure_Init_Additional
: This function initialises WinProp_Additional struct with reasonable values

WinProp_Structure_Init_Model_COST
: This function initialises Model_COST struct with reasonable values

WinProp_Structure_Init_Model_FREESPACE
: This function initialises Model_FREESPACE struct with reasonable values

WinProp_Structure_Init_Model_MOTLEYKEENAN
: This function initialises Model_MOTLEYKEENAN struct with reasonable values

WinProp_Structure_Init_Model_DPM
: This function initialises Model_DPM struct with reasonable values

WinProp_Structure_Init_Model_VRT
: This function initialises Model_VRT struct with reasonable values

WinProp_Structure_Init_VRT_KE_PARA
: This function initialises VRT_KE_PARA struct with initial values

WinProp_Structure_Init_Area
: This function initialises WinProp_Area struct with reasonable values

WinProp_Structure_Init_AreaPlane
: This function initialises WinProp_AreaPlane struct with reasonable values

WinProp_Structure_Init_Pattern
: This function initialises WinProp_Pattern struct with reasonable values

WinProp_Structure_Init_Calib
: This function initialises WinProp_Calib_DPM struct with reasonable values

WinProp_Structure_Init_Coverage_Para
: This function initialises WinProp_Coverage_Para struct with reasonable values

WinProp_Structure_Init_RayMatrix
: This function initialises WinProp_RayMatrix struct with reasonable values

WinProp_Structure_Init_RayMatrixElement
: This function initialises WinProp_RayMatrixElement struct with reasonable values

WinProp_Structure_Init_Ray
: This function initialises WinProp_Ray struct with reasonable values

WinProp_Structure_Init_RayInteraction
: This function initialises WinProp_RayInteraction struct with reasonable values

WinProp_Structure_Init_Result
: This function initialises WinProp_Result struct with reasonable values

WinProp_Structure_Init_ResultPlane
: This function initialises WinProp_ResultPlane struct with reasonable values

WinProp_Structure_Init_ResultPlaneList
: This function initialises WinProp_ResultPlaneList struct with initial values

WinProp_Structure_Init_RayMatrixList
: This function initialises WinProp_RayMatrixList struct with initial values

WinProp_Structure_Init_Receiver
: This function initialises WinProp_Receiver struct with reasonable values

WinProp_Structure_Init_ResultPoint
: This function initialises WinProp_ResultPoint struct with reasonable values

WinProp_Structure_Init_ResultPointsList
: This function initialises WinProp_ResultPointsList struct with reasonable values

WinProp_Pattern_Alloc
: This function allocates memory to the relevant fields of the WinProp_Pattern struct.

WinProp_Pattern_Free
: This function frees memory allocated to WinProp_Pattern.

WinProp_Structure_Init_ParameterHybrid
: This function initialises WinProp_ParaHybrid struct with reasonable values

WinProp_Structure_Init_ParameterRCS
: This function initialises RCS_PARA struct with initial values

WinProp_Structure_Init_ParameterMain
: This function initialises WinProp_ParaMain struct with reasonable values

WinProp_Structure_Init_ParameterRural
: This function initialises WinProp_ParaRural struct with reasonable values

WinProp_Structure_Init_Measurement
: This function initialises WinProp_Measurement struct with reasonable values

WinProp_Structure_Init_MeasurementPoint
: This function initialises WinProp_MeasurementPoint struct with reasonable values

WinProp_Structure_Init_Model_UrbanIRT
: This function initialises Model_UrbanIRT struct with reasonable values

WinProp_Structure_Init_Model_UrbanITU1411
: This function initialises Model_UrbanITU1411 struct with reasonable values

WinProp_Structure_Init_PreProUrban
: This function initialises WinProp_PreProUrban struct with reasonable values

WinProp_Structure_Init_Legend
: This function initialises WinProp_Legend struct with reasonable values

WinProp_Structure_Init_SamplingPoint
: This function initialises WinProp_SamplingPoint struct with reasonable values

WinProp_Structure_Init_Model_RayTracing
: This function initialises Model_RAYTRACING struct with reasonable values

WinProp_Structure_Init_Model_RuralPE
: This function initialises Model_RuralPE struct with reasonable values

WinProp_Structure_Init_Model_RuralITU1546
: This function initialises Model_RuralITU1546 struct with reasonable values

WinProp_Structure_Init_Model_RuralITU526
: This function initialises Model_RuralITU526 struct with reasonable values

WinProp_Structure_Init_Model_RuralITM
: This function initialises Model_RuralITM struct with reasonable values

WinProp_Structure_Init_ResultMonteCarlo
: This function initialises WinProp_ResultMonteCarlo struct with reasonable values

WinProp_Structure_Init_MonteCarloStatistics
: This function initialises WinProp_MonteCarloStatistics struct with reasonable values

WinProp_Structure_Init_TransmitterLoad
: This function initialises WinProp_TransmitterLoad struct with reasonable values

WinProp_Structure_Init_TrajectoryPoint
: This function initialises WinProp_Trajectory_Point struct with reasonable values

WinProp_Structure_Init_TrajectorySample
: This function initialises WinProp_Trajectory_Sample struct with reasonable values

WinProp_Structure_Init_Trajectory
: This function initialises WinProp_Trajectory struct with reasonable values

WinProp_Structure_Init_ResultTrajectory
: This function initialises WinProp_ResultTrajectory struct with reasonable values

WinProp_Structure_Init_ResultTrajectoryList
: This function initialises WinProp_ResultTrajectoryList struct with reasonable values

WinProp_Structure_Init_Additional_Freq_Loss
: This function initialises WinProp_Additional_Freq_Loss struct with initial values

WinProp_Structure_Init_Polygon
: This function initialises WinProp_Polygon struct with initial values

WinProp_Structure_Init_PLANEPOINT
: This function initialises PLANEPOINT struct with initial values

WinProp_Structure_Init_Dynamic_Sets
: This function initialises WinProp_Dynamic_Sets struct with initial values

WinProp_Group_Dynamics_Init
: This function initialises WinProp_Group_Dynamics struct with initial values

WinProp_Structure_Generate_Beams
: This function initialises WinProp_Generate_Beams struct with initial values

WinProp_Structure_Convert_Beams
: This function initialises WinProp_Convert_Beams struct with initial values

WinProp_Structure_WinProp_Antenna_Array
: This function initialises WinProp_Antenna_Array struct with initial values

WinProp_Structure_Init_PrePro
: Initialization of structure WinProp_PrePro with reasonable default values.

Function Details

int ***WinProp_Structure_Init_Callback****(*WinProp_Callback * *ToInit)*

Description

This function initialises callback functions

Parameters

WinProp_Callback * *ToInit*
: structure to initialise.

Returns An integer: 0.

int ***WinProp_Structure_Init_Scenario****(*WinProp_Scenario * *ToInit)*

Description

This function initialises WinProp_Scenario struct with reasonable values

Parameters

WinProp_Scenario * *ToInit*
: struct to initialise.

Returns An integer: 0.

int ***WinProp_Structure_Init_Dynamic_Antenna****(*WinProp_Dynamic_Antenna * *ToInit)*

Description

This function initialises WinProp_Dynamic_Antenna struct with reasonable values

Parameters

WinProp_Dynamic_Antenna * *ToInit*
struct to initialise.

Returns An integer: 0.

int ***WinProp_Structure_Init_Antenna***(WinProp_Antenna * *ToInit*)

Description

This function initialises WinProp_Antenna struct with reasonable values

Parameters

WinProp_Antenna * *ToInit*
struct to initialise.

Returns An integer: 0.

int ***WinProp_Structure_Init_Carrier***(WinProp_Carrier * *ToInit*)

Description

This function initialises WinProp_Carrier struct with reasonable values

Parameters

WinProp_Carrier * *ToInit*
struct to initialise.

Returns An integer: 0.

int ***WinProp_Structure_Init_Propagation_Results***(WinProp_Propagation_Results * *ToInit*)

Description

This function initialises WinProp_Propagation_Results struct with reasonable values

Parameters

WinProp_Propagation_Results * *ToInit*
struct to initialise.

Returns An integer: 0.

int ***WinProp_Structure_Init_Additional***(WinProp_Additional * *ToInit*)

Description

This function initialises WinProp_Additional struct with reasonable values

Parameters

WinProp_Additional * *ToInit*
struct to initialise.

Returns An integer: 0.

int ***WinProp_Structure_Init_Model_COST***(Model_COST * *ToInit*)

Description

This function initialises Model_COST struct with reasonable values

Parameters

Model_COST * *ToInit*
struct to initialise.

Returns An integer: 0.

int ***WinProp_Structure_Init_Model_FREESPACE***(Model_FREESPACE * *ToInit*)

Description

This function initialises Model_FREESPACE struct with reasonable values

Parameters

Model_FREESPACE * *ToInit*
struct to initialise.

Returns An integer: 0.

int ***WinProp_Structure_Init_Model_MOTLEYKEENAN***(Model_MOTLEYKEENAN * *ToInit*)

Description

This function initialises Model_MOTLEYKEENAN struct with reasonable values

Parameters

Model_MOTLEYKEENAN * *ToInit*
struct to initialise.

Returns An integer: 0.

int ***WinProp_Structure_Init_Model_DPM***(Model_DPM * *ToInit*)

Description

This function initialises Model_DPM struct with reasonable values

Parameters

Model_DPM * *ToInit*
struct to initialise.

Returns An integer: 0.

int ***WinProp_Structure_Init_Model_VRT***(Model_VRT * *ToInit*)

Description

This function initialises Model_VRT struct with reasonable values

Parameters

Model_VRT * *ToInit*
struct to initialise.

Returns An integer: 0.

int ***WinProp_Structure_Init_VRT_KE_PARA***(VRT_KE_PARA * *ToInit*)

Description

This function initialises VRT_KE_PARA struct with initial values

Parameters

VRT_KE_PARA * *ToInit*
struct to initialise.

Returns An integer: 0.

int ***WinProp_Structure_Init_Area****(WinProp_Area * ToInit)*

Description

This function initialises WinProp_Area struct with reasonable values

Parameters

WinProp_Area * *ToInit*
struct to initialise.

Returns An integer: 0.

int ***WinProp_Structure_Init_AreaPlane****(WinProp_AreaPlane * ToInit)*

Description

This function initialises WinProp_AreaPlane struct with reasonable values

Parameters

WinProp_AreaPlane * *ToInit*
struct to initialise.

Returns An integer: 0.

int ***WinProp_Structure_Init_Pattern****(WinProp_Pattern * ToInit)*

Description

This function initialises WinProp_Pattern struct with reasonable values

Parameters

WinProp_Pattern * *ToInit*
struct to initialise.

Returns An integer: 0.

int ***WinProp_Structure_Init_Calib****(WinProp_Calib_DPM * ToInit)*

Description

This function initialises WinProp_Calib_DPM struct with reasonable values

Parameters

WinProp_Calib_DPM * *ToInit*
struct to initialise.

Returns An integer: 0.

int ***WinProp_Structure_Init_Coverage_Para****(WinProp_Coverage_Para * ToInit)*

Description

This function initialises WinProp_Coverage_Para struct with reasonable values

Parameters

WinProp_Coverage_Para * *ToInit*
struct to initialise.

Returns An integer: 0.

int ***WinProp_Structure_Init_RayMatrix***(WinProp_RayMatrix * *ToInit)*

Description

This function initialises WinProp_RayMatrix struct with reasonable values

Parameters

WinProp_RayMatrix * *ToInit*
struct to initialize.

Returns An integer: 0.

int ***WinProp_Structure_Init_RayMatrixElement***(WinProp_RayMatrixElement * *ToInit)*

Description

This function initialises WinProp_RayMatrixElement struct with reasonable values

Parameters

WinProp_RayMatrixElement * *ToInit*
struct to initialize.

Returns An integer: 0.

int ***WinProp_Structure_Init_Ray***(WinProp_Ray * *ToInit)*

Description

This function initialises WinProp_Ray struct with reasonable values

Parameters

WinProp_Ray * *ToInit*
struct to initialize.

Returns An integer: 0.

int ***WinProp_Structure_Init_RayInteraction***(WinProp_RayInteraction * *ToInit)*

Description

This function initialises WinProp_RayInteraction struct with reasonable values

Parameters

WinProp_RayInteraction * *ToInit*
struct to initialize.

Returns An integer: 0.

int ***WinProp_Structure_Init_Result***(WinProp_Result * *ToInit)*

Description

This function initialises WinProp_Result struct with reasonable values

Parameters

WinProp_Result * *ToInit*
struct to initialise.

Returns An integer: 0.

int ***WinProp_Structure_Init_ResultPlane***(WinProp_ResultPlane * *ToInit)*

Description

This function initialises WinProp_ResultPlane struct with reasonable values

Parameters

WinProp_ResultPlane * *ToInit*
struct to initialise.

Returns An integer: 0.

int ***WinProp_Structure_Init_ResultPlaneList***(WinProp_ResultPlaneList * *ToInit)*

Description

This function initialises WinProp_ResultPlaneList struct with initial values

Parameters

WinProp_ResultPlaneList * *ToInit*
struct to initialise.

Returns An integer: 0.

int ***WinProp_Structure_Init_RayMatrixList***(WinProp_RayMatrixList * *ToInit)*

Description

This function initialises WinProp_RayMatrixList struct with initial values

Parameters

WinProp_RayMatrixList * *ToInit*
struct to initialise.

Returns An integer: 0.

int ***WinProp_Structure_Init_Receiver***(WinProp_Receiver * *ToInit)*

Description

This function initialises WinProp_Receiver struct with reasonable values

Parameters

WinProp_Receiver * *ToInit*
struct to initialise.

Returns An integer: 0.

int ***WinProp_Structure_Init_ResultPoint***(WinProp_ResultPoint * *ToInit)*

Description

This function initialises WinProp_ResultPoint struct with reasonable values

Parameters

WinProp_ResultPoint * *ToInit*
struct to initialise.

Returns An integer: 0.

int ***WinProp_Structure_Init_ResultPointsList***(WinProp_ResultPointsList * *ToInit*)

Description

This function initialises WinProp_ResultPointsList struct with reasonable values

Parameters

WinProp_ResultPointsList * *ToInit*
struct to initialise.

Returns An integer: 0.

int ***WinProp_Pattern_Alloc***(WinProp_Pattern * *ToInit*)

Description

This function allocates memory to the relevant fields of the WinProp_Pattern struct.

Parameters

WinProp_Pattern * *ToInit*
WinProp_Pattern struct.

Returns An integer: 0.

int ***WinProp_Pattern_Free***(WinProp_Pattern * *ToFree*)

Description

This function frees memory allocated to WinProp_Pattern.

Parameters

WinProp_Pattern * *ToFree*
struct to free.

Returns An integer: 0.

int ***WinProp_Structure_Init_ParameterHybrid***(WinProp_ParaHybrid * *Para*)

Description

This function initialises WinProp_ParaHybrid struct with reasonable values

Parameters

WinProp_ParaHybrid * *Para*
struct to initialise.

Returns An integer: 0.

int ***WinProp_Structure_Init_ParameterRCS***(RCS_PARA * *Para*)

Description

This function initialises RCS_PARA struct with initial values

Parameters

RCS_PARA * *Para*
struct to initialise.

Returns An integer: 0.

int ***WinProp_Structure_Init_ParameterMain***(WinProp_ParaMain * *ParameterUrban*)

Description

This function initialises WinProp_ParaMain struct with reasonable values

Parameters

WinProp_ParaMain * *ParameterUrban*
struct to initialise.

Returns An integer: 0.

int ***WinProp_Structure_Init_ParameterRural***(WinProp_ParaRural * *Para*)

Description

This function initialises WinProp_ParaRural struct with reasonable values

Parameters

WinProp_ParaRural * *Para*
struct to initialise.

Returns An integer: 0.

int ***WinProp_Structure_Init_Measurement***(WinProp_Measurement * *Para*)

Description

This function initialises WinProp_Measurement struct with reasonable values

Parameters

WinProp_Measurement * *Para*
struct to initialise.

Returns An integer: 0.

int ***WinProp_Structure_Init_MeasurementPoint***(WinProp_MeasurementPoint * *Point*)

Description

This function initialises WinProp_MeasurementPoint struct with reasonable values

Parameters

WinProp_MeasurementPoint * *Point*
struct to initialise.

Returns An integer: 0.

int ***WinProp_Structure_Init_Model_UrbanIRT***(Model_UrbanIRT * *ParaIRT*)

Description

This function initialises Model_UrbanIRT struct with reasonable values

Parameters

Model_UrbanIRT * *ParaIRT*
struct to initialise.

Returns An integer: 0.

int ***WinProp_Structure_Init_Model_UrbanITU1411***(Model_UrbanITU1411 * *ToInit*)

Description

This function initialises Model_UrbanITU1411 struct with reasonable values

Parameters

Model_UrbanITU1411 * *ToInit*
struct to initialise.

Returns An integer: 0.

void ***WinProp_Structure_Init_PreProUrban***(WinProp_PreProUrban * *Init*)

Description

This function initialises WinProp_PreProUrban struct with reasonable values

Parameters

WinProp_PreProUrban * *Init*
struct to initialise.

Returns None

int ***WinProp_Structure_Init_Legend***(WinProp_Legend * *ToInit*)

Description

This function initialises WinProp_Legend struct with reasonable values

Parameters

WinProp_Legend * *ToInit*
struct to initialise.

Returns An integer: 0.

int ***WinProp_Structure_Init_SamplingPoint***(WinProp_SamplingPoint * *ToInit*)

Description

This function initialises WinProp_SamplingPoint struct with reasonable values

Parameters

WinProp_SamplingPoint * *ToInit*
struct to initialise.

Returns An integer: 0.

int ***WinProp_Structure_Init_Model_RayTracing***(Model_RAYTRACING * *ToInit*)

Description

This function initialises Model_RAYTRACING struct with reasonable values

Parameters

Model_RAYTRACING * *ToInit*
struct to initialise.

Returns An integer: 0.

int ***WinProp_Structure_Init_Model_RuralPE***(Model_RuralPE * *ToInit*)

Description

This function initialises Model_RuralPE struct with reasonable values

Parameters

Model_RuralPE * *ToInit*
struct to initialise.

Returns An integer: 0.

int ***WinProp_Structure_Init_Model_RuralITU1546***(Model_RuralITU1546 * *ToInit*)

Description

This function initialises Model_RuralITU1546 struct with reasonable values

Parameters

Model_RuralITU1546 * *ToInit*
struct to initialise.

Returns An integer: 0.

int ***WinProp_Structure_Init_Model_RuralITU526***(Model_RuralITU526 * *ToInit*)

Description

This function initialises Model_RuralITU526 struct with reasonable values

Parameters

Model_RuralITU526 * *ToInit*
struct to initialise.

Returns An integer: 0.

int ***WinProp_Structure_Init_Model_RuralITM***(Model_RuralITM * *ToInit*)

Description

This function initialises Model_RuralITM struct with reasonable values

Parameters

Model_RuralITM * *ToInit*
struct to initialise.

Returns An integer: 0.

void ***WinProp_Structure_Init_ResultMonteCarlo***(WinProp_ResultMonteCarlo * *ToInit*)

Description

This function initialises WinProp_ResultMonteCarlo struct with reasonable values

Parameters

WinProp_ResultMonteCarlo * *ToInit*
struct to initialise.

Returns None

void ***WinProp_Structure_Init_MonteCarloStatistics***(WinProp_MonteCarloStatistics * *ToInit*)

Description

This function initialises WinProp_MonteCarloStatistics struct with reasonable values

Parameters

WinProp_MonteCarloStatistics * *ToInit*
struct to initialise.

Returns None

void ***WinProp_Structure_Init_TransmitterLoad***(WinProp_TransmitterLoad * *ToInit*)

Description

This function initialises WinProp_TransmitterLoad struct with reasonable values

Parameters

WinProp_TransmitterLoad * *ToInit*
struct to initialise.

Returns None

void ***WinProp_Structure_Init_TrajectoryPoint***(WinProp_Trajectory_Point * *ToInit*)

Description

This function initialises WinProp_Trajectory_Point struct with reasonable values

Parameters

WinProp_Trajectory_Point * *ToInit*
struct to initialise.

Returns None

void ***WinProp_Structure_Init_TrajectorySample***(WinProp_Trajectory_Sample * *ToInit*)

Description

This function initialises WinProp_Trajectory_Sample struct with reasonable values

Parameters

WinProp_Trajectory_Sample * *ToInit*
struct to initialise.

Returns None

void ***WinProp_Structure_Init_Trajectory***(WinProp_Trajectory * *ToInit*)

Description

This function initialises WinProp_Trajectory struct with reasonable values

Parameters

WinProp_Trajectory * *ToInit*
struct to initialise.

Returns None

void ***WinProp_Structure_Init_ResultTrajectory***(WinProp_ResultTrajectory * *ToInit*)

Description

This function initialises WinProp_ResultTrajectory struct with reasonable values

Parameters

WinProp_ResultTrajectory * *ToInit*
struct to initialise.

Returns None

void ***WinProp_Structure_Init_ResultTrajectoryList***(WinProp_ResultTrajectoryList * *ToInit*)

Description

This function initialises WinProp_ResultTrajectoryList struct with reasonable values

Parameters

WinProp_ResultTrajectoryList * *ToInit*
struct to initialise.

Returns None

void ***WinProp_Structure_Init_Additional_Freq_Loss***(WinProp_Additional_Freq_Loss * *ToInit*)

Description

This function initialises WinProp_Additional_Freq_Loss struct with initial values

Parameters

WinProp_Additional_Freq_Loss * *ToInit*
struct to initialise.

Returns None

void ***WinProp_Structure_Init_Polygon***(WinProp_Polygon * *ToInit*)

Description

This function initialises WinProp_Polygon struct with initial values

Parameters

WinProp_Polygon * *ToInit*
struct to initialise.

Returns None

void ***WinProp_Structure_Init_PLANEPOINT***(PLANEPOINT * *ToInit*)

Description

This function initialises PLANEPOINT struct with initial values

Parameters

PLANEPOINT * *ToInit*
struct to initialise.

Returns None

void ***WinProp_Structure_Init_Dynamic_Sets***(WinProp_Dynamic_Sets * *ToInit*)

Description

This function initialises WinProp_Dynamic_Sets struct with initial values

Parameters

WinProp_Dynamic_Sets * *ToInit*
struct to initialise.

Returns None

void ***WinProp_Group_Dynamics_Init***(WinProp_Group_Dynamics * *ToInit*)

Description

This function initialises WinProp_Group_Dynamics struct with initial values

Parameters

WinProp_Group_Dynamics * *ToInit*
struct to initialise.

Returns None

int ***WinProp_Structure_Generate_Beams***(WinProp_Generate_Beams * *ToInit*)

Description

This function initialises WinProp_Generate_Beams struct with initial values

Parameters

WinProp_Generate_Beams * *ToInit*
struct to initialise.

Returns None

int ***WinProp_Structure_Convert_Beams***(WinProp_Convert_Beams * *ToInit*)

Description

This function initialises WinProp_Convert_Beams struct with initial values

Parameters

WinProp_Convert_Beams * *ToInit*
struct to initialise.

Returns None

int ***WinProp_Structure_WinProp_Antenna_Array***(WinProp_Antenna_Array * *ToInit*)

Description

This function initialises WinProp_Antenna_Array struct with initial values

Parameters

WinProp_Antenna_Array * *ToInit*
struct to initialise.

Returns None

int ***WinProp_Structure_Init_PrePro****(*WinProp_PrePro * *ToInit)*

Description

Initialization of structure WinProp_PrePro with reasonable default values.

Parameters

WinProp_PrePro * *ToInit*
Structure to be initialized (see WinProp_PrePro).

Returns An integer: 0 = success, failure otherwise.

The documentation was generated from the following file:

- `source/Interface/Init.h`

2.2.7 Functions for database pre-processing

Function List

OutdoorPlugIn_ComputePrePro
Computation of preprocessing for an urban vector database.

WinProp_PreProcessIndoor
This functions starts the preprocessing for the 3D Intelligent Ray Tracing. The vector database has to be passed to the function.

Function Details

int ***OutdoorPlugIn_ComputePrePro****(const* WinProp_PreProUrban * *ParameterPrePro, const char * OutputFileName,* URBAN_BUILDINGS * *Buildings,* TOPOGRAPHY * *Topography, const* WinProp_Callback * *Callback, const void * ForTheFuture)*

Description

Computation of preprocessing for an urban vector database.

Parameters

const WinProp_PreProUrban * *ParameterPrePro*
Configuration of preprocessing (see WinProp_PreProUrban).

*const char * OutputFileName*
Filename of preprocessed vector database. File suffix will be added automatically.

URBAN_BUILDINGS * *Buildings*
: Vector buildings (see URBAN_BUILDINGS) used for preprocessing. Instead of using this structure the filename of a WinProp vector database can be defined in the parameters ParameterPrePro.

TOPOGRAPHY * *Topography*
: Topography (see TOPOGRAPHY) used for preprocessing. Instead of using this structure the filename of a WinProp topography database can be defined in the parameters ParameterPrePro.

const WinProp_Callback * *Callback*
: Callback functions for progress output (see WinProp_Callback).

const void * *ForTheFuture*
: Not yet supported (set to NULL).

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_PreProcessIndoor****(const char * SourceVectordatabase, const char * DestinationVectordatabase, const* WinProp_PrePro * *Para, const* WinProp_Callback * *Callback)*

Description

This functions starts the preprocessing for the 3D Intelligent Ray Tracing. The vector database has to be passed to the function.

Parameters

const char * *SourceVectordatabase*
: Vector database for the preprocessing (both *.ida and *.idb files are supported).

const char * *DestinationVectordatabase*
: Filename of destination file (*.idi). The suffix must not be passed to the function.

const WinProp_PrePro * *Para*
: Parameters for the preprocessing (see WinProp_PrePro).

const WinProp_Callback * *Callback*
: Pointers to callback functions (see WinProp_Callback).

Returns An integer: 0 = success, failure otherwise.

The documentation was generated from the following file:

- `source/Interface/OutdoorPlugInPrePro.h`

2.2.8 Functions for computing total power

Function List

WinProp_Total_Power
: Computation of the power superposition for a set of power results

WinProp_Total_Power_Points
: Computation of the power superposition for a set of prediction point power results

WinProp_Total_Power_Trajectories
: Computation of the power superposition for a set of prediction trajectory power results

WinProp_Total_Power_Planes
: Computation of the power superposition for a set of prediction plane power results

Function Details

int ***WinProp_Total_Power****(const int NrResults, const* WinProp_Result * *PowerResults, const* WinProp_RayMatrix * *RayMatrices, const* TOTAL_POWER_MODE *Mode, const float Frequency, const* WinProp_Callback * *WinPropCallback,* WinProp_Result * *TotalPower)*

Description

Computation of the power superposition for a set of power results

Parameters

const int NrResults
: The number of results to superpose.

const WinProp_Result * *PowerResults*
: The power results to superpose. This is an array of size equal to NrResults.

const WinProp_RayMatrix * *RayMatrices*
: The ray matrices to superpose. This is an array of size equal to NrResults. Only relevant if mode = TOTAL_POWER_MODE::COHERENT.

const TOTAL_POWER_MODE *Mode*
: The power superposition mode see TOTAL_POWER_MODE.

const float Frequency
: The frequency of the results in [MHz].

const WinProp_Callback * *WinPropCallback*
: The configuration of callback functions (see WinProp_Callback).

WinProp_Result * *TotalPower*
: If non-null, the total power for the input results.

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_Total_Power_Points****(const int NrResults, const* WinProp_ResultPointsList * *PointLists, const* TOTAL_POWER_MODE *Mode, const float Frequency, const* WinProp_Callback * *WinPropCallback,* WinProp_ResultPointsList * *TotalPowerPoints)*

Description

Computation of the power superposition for a set of prediction point power results

Parameters

const int NrResults
: The number of results to superpose.

const WinProp_ResultPointsList * *PointLists*
The point lists results to superpose. This is an array of size equal to NrResults.

const TOTAL_POWER_MODE *Mode*
The power superposition mode see TOTAL_POWER_MODE.

const float Frequency
The frequency of the results in [MHz].

const WinProp_Callback * *WinPropCallback*
The configuration of callback functions (see WinProp_Callback).

WinProp_ResultPointsList * *TotalPowerPoints*
If non-null, the total power for the input point lists.

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_Total_Power_Trajectories****(const int NrResults, const* WinProp_ResultTrajectoryList * *TrajectoryLists, const* TOTAL_POWER_MODE *Mode, const float Frequency, const* WinProp_Callback * *WinPropCallback,* WinProp_ResultTrajectoryList * *TotalPowerTrajectories)*

Description

Computation of the power superposition for a set of prediction trajectory power results

Parameters

const int NrResults
The number of results to superpose.

const WinProp_ResultTrajectoryList * *TrajectoryLists*
The trajectory lists results to superpose. This is an array of size equal to NrResults.

const TOTAL_POWER_MODE *Mode*
The power superposition mode see TOTAL_POWER_MODE.

const float Frequency
The frequency of the results in [MHz].

const WinProp_Callback * *WinPropCallback*
The configuration of callback functions (see WinProp_Callback).

WinProp_ResultTrajectoryList * *TotalPowerTrajectories*
If non-null, the total power for the input trajectory lists.

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_Total_Power_Planes****(const int NrResults, const* WinProp_ResultPlaneList * *PowerResultPlanes, const* WinProp_RayMatrixList * *RayMatrixPlanes, const* TOTAL_POWER_MODE *Mode, const float Frequency, const* WinProp_Callback * *WinPropCallback,* WinProp_ResultPlaneList * *TotalPowerPlanes)*

Description

Computation of the power superposition for a set of prediction plane power results

Parameters

const int NrResults
The number of results to superpose.

const WinProp_ResultPlaneList * *PowerResultPlanes*
The power results to superpose. This is an array of size equal to NrResults.

const WinProp_RayMatrixList * *RayMatrixPlanes*
The ray matrices to superpose. This is an array of size equal to NrResults. Only relevant if mode = TOTAL_POWER_MODE::COHERENT.

const TOTAL_POWER_MODE *Mode*
The power superposition mode see TOTAL_POWER_MODE.

const float Frequency
The frequency of the results in [MHz].

const WinProp_Callback * *WinPropCallback*
The configuration of callback functions (see WinProp_Callback).

WinProp_ResultPlaneList * *TotalPowerPlanes*
If non-null, the total power for the input plane results.

Returns An integer: 0 = success, failure otherwise.

The documentation was generated from the following file:

- `source/Interface/EngineTotalPower.h`

2.2.9 Outdoor Wave Propagation

Prediction of wave propagation in outdoor (urban/rural) scenarios.

This chapter contains the following:

1. Functions
2. Defines and constants

2.2.10 Functions

Function List

OutdoorPlugIn_ComputePrediction
Computation of wave propagation prediction for a specified scenario. All relevant information has to be passed to the function.

OutdoorPlugIn_ComputePoints
Computation of wave propagation prediction for a specified scenario and the specified individual points. All relevant information has to be passed to the function.

OutdoorPlugIn_ComputeTrajectories
: Computation of wave propagation prediction for a specified scenario and the specified trajectories. All relevant information has to be passed to the function.

OutdoorPlugIn_GetDefaultValue
: This function returns the default maximum pathloss value in an outdoor scenario.

OutdoorPlugIn_FreePredictionArea
: This functions frees allocated memory

OutdoorPlugIn_FreePredictionHeights
: This functions frees allocated memory

OutdoorPlugIn_PointInsidePolygon
: This function checks whether a point lies inside polygon

OutdoorPlugIn_LineIntersection
: This function computes the intersection between trajectories (lines).

OutdoorPlugIn_ReadPredictionArea
: This function reads in the specified parameters of the prediction area

Function Details

int ***OutdoorPlugIn_ComputePrediction****(const* WinProp_Antenna * *ParameterAntenna,* WinProp_ParaMain * *ParameterUrban, const* WinProp_Receiver * *receivingPoints, const int nrReceivingPoints, const void * ParameterModelUrban,* WinProp_Measurement * *ParameterMeasurements, const* WinProp_ParaRural * *ParameterRural, const* WinProp_ParaHybrid * *ParameterCNP, const* WinProp_Callback * *Callback,* WinProp_Result * *Resultmatrix,* WinProp_RayMatrix * *DataRaysOut,* WinProp_Result * *LOSmatrix,* WinProp_ResultPlaneList * *ResultPlanes,* WinProp_RayMatrixList * *ResultRayMatrix)*

Description

Computation of wave propagation prediction for a specified scenario. All relevant information has to be passed to the function.

Parameters

const WinProp_Antenna * *ParameterAntenna*
: Configuration of antenna (see WinProp_Antenna).

WinProp_ParaMain * *ParameterUrban*
: Configuration of scenario (see WinProp_ParaMain).

const WinProp_Receiver * *receivingPoints*
: Non-null, array of receiving points, of length equal to nrReceivingPoints.

const int nrReceivingPoints
: The number receiving points in receivingPoints.

*const void * ParameterModelUrban*
: Configuration of the urban prediction model. This parameter is optional and depends on the selected prediction model:

- Urban Dominant Path Model (see Model_DPM).

- 3D Intelligent Ray Tracing (see Model_UrbanIRT).

WinProp_Measurement * *ParameterMeasurements*
: Measurement data for calibration of prediction model (see WinProp_Measurement). This parameter is optional.

const WinProp_ParaRural * *ParameterRural*
: Configuration of rural prediction model. Only relevant if a rural prediction is done (see WinProp_ParaRural).

const WinProp_ParaHybrid * *ParameterCNP*
: Configuration of CNP mode. This is not yet supported.

const WinProp_Callback * *Callback*
: Configuration of callback functions (see WinProp_Callback). This parameter is optional.

WinProp_Result * *Resultmatrix*
: If non-null, the result matrix, gets allocated during computation, it is not needed to call WinProp_AllocateResult prior to the computation.

WinProp_RayMatrix * *DataRaysOut*
: If non-null, the ray matrix, gets allocated during computation.

WinProp_Result * *LOSmatrix*
: If non-null, the LOS matrix, gets allocated during computation, it is not needed to call WinProp_AllocateResult prior to the computation.

WinProp_ResultPlaneList * *ResultPlanes*
: If non-null, the ResultPlanes, gets allocated during computation, it is not needed to call WinProp_Structure_Init_ResultPlaneList prior to the computation.

WinProp_RayMatrixList * *ResultRayMatrix*
: If non-null, the result ResultRayMatrix gets allocated during computation, it is not needed to call WinProp_Structure_Init_RayMatrixList prior to the computation.

Returns An integer: 0 = success, failure otherwise.

int **`OutdoorPlugIn_ComputePoints`**(*const* WinProp_Antenna * *ParameterAntenna, *WinProp_ParaMain * *ParameterUrban, const void * ParameterModelUrban, const* WinProp_ParaRural * *ParameterRural, const* WinProp_Receiver * *receivingPoints, const int nrReceivingPoints, const* WinProp_Callback * *Callback,* WinProp_ResultPointsList * *resultPoints)*

Description

Computation of wave propagation prediction for a specified scenario and the specified individual points. All relevant information has to be passed to the function.

Parameters

const WinProp_Antenna * *ParameterAntenna*
: Configuration of antenna (see WinProp_Antenna).

WinProp_ParaMain * *ParameterUrban*
: Configuration of scenario (see WinProp_ParaMain).

*const void * ParameterModelUrban*
Configuration of the urban prediction model. This parameter is optional and depends on the selected prediction model:

- Urban Dominant Path Model (see Model_DPM).
- 3D Intelligent Ray Tracing (see Model_UrbanIRT).

const WinProp_ParaRural * *ParameterRural*
Configuration of rural prediction model. Only relevant if a rural prediction is done (see WinProp_ParaRural).

const WinProp_Receiver * *receivingPoints*
Non-null, array of receiving points, of length equal to nrReceivingPoints.

const int nrReceivingPoints
The number receiving points in receivingPoints.

const WinProp_Callback * *Callback*
Configuration of callback functions (see WinProp_Callback). This parameter is optional.

WinProp_ResultPointsList * *resultPoints*
Non-null, structure containing the points results.

Returns An integer: 0 = success, otherwise an error.

int ***OutdoorPlugIn_ComputeTrajectories****(const* WinProp_Antenna * *ParameterAntenna,* WinProp_ParaMain * *ParameterUrban, const void * ParameterModelUrban, const* WinProp_ParaRural * *ParameterRural, const* WinProp_Trajectory * *trajectories, const int nrTrajectories, const* WinProp_Callback * *Callback,* WinProp_ResultTrajectoryList * *resultTrajectories)*

Description

Computation of wave propagation prediction for a specified scenario and the specified trajectories. All relevant information has to be passed to the function.

Parameters

const WinProp_Antenna * *ParameterAntenna*
Configuration of antenna (see WinProp_Antenna).

WinProp_ParaMain * *ParameterUrban*
Configuration of scenario (see WinProp_ParaMain).

*const void * ParameterModelUrban*
Configuration of the urban prediction model. This parameter is optional and depends on the selected prediction model:

- Urban Dominant Path Model (see Model_DPM).
- 3D Intelligent Ray Tracing (see Model_UrbanIRT).

const WinProp_ParaRural * *ParameterRural*
Configuration of rural prediction model. Only relevant if a rural prediction is done (see WinProp_ParaRural).

const WinProp_Trajectory * *trajectories*
The trajectories, an array of size nrTrajectories.

const int nrTrajectories
The number of trajectories.

const WinProp_Callback * *Callback*
Configuration of callback functions (see WinProp_Callback). This parameter is optional.

WinProp_ResultTrajectoryList * *resultTrajectories*
Non-null, structure containing the trajectory results.

Returns An int.

int ***OutdoorPlugIn_GetDefaultValue****(int Parameter, double * ReturnValue)*

Description

This function returns the default maximum pathloss value in an outdoor scenario.

Parameters

int Parameter
This parameter is defined by the macro INTERMEDIATE_DEFAULT_VALUE_MAXPATHLOSS.

*double * ReturnValue*
The maximum pathloss value.

Returns An integer: 0 = success, failure otherwise.

int ***OutdoorPlugIn_FreePredictionArea****(*COORDPOINT ** *Polygon)*

Description

This functions frees allocated memory

Parameters

COORDPOINT ** *Polygon*
Matrix with coordinate points.

Returns An integer: 0 = success, failure otherwise.

int ***OutdoorPlugIn_FreePredictionHeights****(double ** Heights)*

Description

This functions frees allocated memory

Parameters

*double ** Heights*
Matrix with prediction heights.

Returns An integer: 0 = success, failure otherwise.

int ***OutdoorPlugIn_PointInsidePolygon****(int NrCorners, const* COORDPOINT * *Corners,* COORDPOINT *Point, int Projection, int * Success)*

Description

This function checks whether a point lies inside polygon

Parameters

int NrCorners
Number of corners of the polygon.

const COORDPOINT * *Corners*
Coordinates of the polygon's corner points.

COORDPOINT *Point*
Coordinates of the test point.

int Projection
Projection plane:

- 1 = Projection in the Y-Z plane.
- 2 = Projection in the X-Z plane.
- 3 = Projection in the X-Y plane.

*int * Success*
Indicates success (1) or failure (0) of the inner computations.

Returns An integer: 1 = the point lies in the polygon, 0 otherwise.

int ***OutdoorPlugIn_LineIntersection****(*COORDPOINT *Line1Point1,* COORDPOINT *Line1Point2,* COORDPOINT *Line2Point1,* COORDPOINT *Line2Point2,* COORDPOINT * *IntersectionPoint)*

Description

This function computes the intersection between trajectories (lines).

Parameters

COORDPOINT *Line1Point1*
Starting point of Line 1.

COORDPOINT *Line1Point2*
End point of Line 1.

COORDPOINT *Line2Point1*
Starting point of Line 2.

COORDPOINT *Line2Point2*
End point of Line 2.

COORDPOINT * *IntersectionPoint*
The intersection point between the two lines.

Returns An integer:

int ***OutdoorPlugIn_ReadPredictionArea****(const char * FileName, int * NumberCorners,* COORDPOINT *** Polygon,* WINPROP_DATABASE_TYPE_URBAN *DatabaseFileType, double * Resolution, int * NumberHeights, double ** Heights, int * PolygonUsed)*

Description

This function reads in the specified parameters of the prediction area

Parameters

*const char * FileName*
Name of database.

*int * NumberCorners*
Number of corners of the polygon.

COORDPOINT ** *Polygon*
Coordinates of the polygon's corner points.

WINPROP_DATABASE_TYPE_URBAN *DatabaseFileType*
Type of the database file.

*double * Resolution*
Resolution.

*int * NumberHeights*
Number of prediction heights.

*double ** Heights*
Prediction heights.

*int * PolygonUsed*
Pointer to an integer that specifies if a polygon is used (1) or not (0).

Returns An int.

The documentation was generated from the following file:

- `source/Interface/OutdoorPlugIn.h`

2.2.11 Defines and constants

Detailed Description

Defines and constants

Clutter computation mode.

WINPROP_CLUTTER_HEIGHT_MODE

- CLUTTERLOSS_HEIGHTS_NONE : No clutter heights considered. See WinProp_ParaMain::ClutterConsideration where this is used.
- CLUTTERLOSS_HEIGHTS : Constant height for all clutter. See WinProp_ParaMain::ClutterConsideration where this is used.
- CLUTTERLOSS_HEIGHTS_STATISTIC : Heights according to a statistical distribution. See WinProp_ParaMain::ClutterConsideration where this is used.

WINPROP_CLUTTER_ATTENUATION_MODE

- CLUTTERATTENUATION_NONE : no attenuation.
- CLUTTERATTENUATION_LOCAL : only local class at receiver location.
- CLUTTERATTENUATION_RAY_TOTAL : dominant class along ray, constant weight factor.
- CLUTTERATTENUATION_RAY_DECREASE : dominant class along ray, decreasing weight factor.

Error codes and Parameters associated with a .mif database.

MAPINFO_FILENAME_ADD *"_area"*
Suffix.

MAPINFO_ERROR_FILENOTFOUND *1*
Error code for: .mif file not found.

MAPINFO_ERROR_NRPOLYGONS *2*
Error code for: incorrect number of polygons in .mif file.

MAPINFO_ERROR_POLYGONNOTVALID *3*
Error code for: invalid polygon in .mif file.

MAPINFO_ERROR_FEATURE *4*
Error code for: incorrect element type in .mif file.

MAPINFO_ERROR_PARAMETERWRONG *5*
Error code for: unknown format in .mif file.

Selection of type of database.

WINPROP_ERROR_DATABASE *6*
Error code for: error while reading database.

WINPROP_ERROR_LICENSE *7*
Error code for: license error.

WINPROP_ERROR_WRONGPARAMETER *8*
Error code for: wrong parameter specified.

Filter results mode.

FILTERMODE_MEAN *0*
Mean based result filtering. See WinProp_ParaMain::FilterMode where this is used.

FILTERMODE_MEDIAN *1*
Median based result filtering. See WinProp_ParaMain::FilterMode where this is used.

Default values for different parameters

INTERMEDIATE_DEFAULT_VALUE_MAXPATHLOSS *1*
Use intermediate value for maximum path loss.

Enum

WINPROP_DATABASE_TYPE_URBAN

- DATABASE_TYPE_MAPINFO : The database is a .mif file. This is used in the DatabaseFileType parameter of the OutdoorPlugIn_ReadPredictionArea function.
- DATABASE_TYPE_WINPROP_ODB : The database is a .odb file. This is used in the DatabaseFileType parameter of the OutdoorPlugIn_ReadPredictionArea function.
- DATABASE_TYPE_WINPROP_OPB : The database is a .opb file. This is used in the DatabaseFileType parameter of the OutdoorPlugIn_ReadPredictionArea function.

- DATABASE_TYPE_WINPROP_OCB : The database is a .ocb file. This is used in the DatabaseFileType parameter of the OutdoorPlugIn_ReadPredictionArea function.
- DATABASE_TYPE_WINPROP_OIB : The database is a .oib file. This is used in the DatabaseFileType parameter of the OutdoorPlugIn_ReadPredictionArea function.

PREDMODEL_URBAN

- PREDMODEL_COST : COST 231 Walfisch-Ikegami model used in an urban scenario.
- PREDMODEL_UDP : Urban dominant path model.
- PREDMODEL_IRT : 3D intelligent ray tracing model.
- PREDMODEL_KE : Knife edge model.
- PREDMODEL_ITU1411 : ITU-R P.1411 model used in an urban scenario.

PREDMODEL_RURAL

- PREDMODEL_RURAL_NONE : No rural prediction required (only necessary in case of multi scenario predictions).
- PREDMODEL_RURAL_HATA : Okumura-Hata model.
- PREDMODEL_RURAL_2RAY_EMP : Empirical two ray model.
- PREDMODEL_RURAL_2RAY_DET : Deterministic two ray model.
- PREDMODEL_RURAL_RDP : Rural dominant path model.
- PREDMODEL_RURAL_PE : Rural parabolic equation model.
- PREDMODEL_RURAL_ITU1546 : Rural ITU-R P.1546 broadcasting model.
- PREDMODEL_RURAL_ITU526 : Rural ITU-R P.526 model.
- PREDMODEL_RURAL_ITM : Rural irregular terrain model (ITM, Longley-Rice).

WINPROP_OUTDOOR_SCENARIO_MODE

- SCENARIOMODE_RURAL : Rural prediction based on topography and optionally clutter data. See WinProp_ParaMain::ScenarioMode where this is used.
- SCENARIOMODE_URBAN : Urban prediction based on topography and vector building data. See WinProp_ParaMain::ScenarioMode where this is used.

WINPROP_BUILDINGS_MODE

- BUILDINGSMODE_NONE : No buildings will be considered.
- BUILDINGSMODE_RAM : Load buildings from RAM.
- BUILDINGSMODE_UDP : Load OPB database. Only possible with the urban dominant path model (see PREDMODEL_UDP in WinProp_ParaMain::PredictionModelUrban).
- BUILDINGSMODE_IRT : Load OIB database. Only possible with the IRT model (see PREDMODEL_IRT in WinProp_ParaMain::PredictionModelUrban).
- BUILDINGSMODE_COST : Load OCB database. Only possible with COST 231 Walfisch-Ikegami Model (see PREDMODEL_COST in WinProp_ParaMain::PredictionModelUrban).
- BUILDINGSMODE_BINARY : Load buildings from binary WinProp file (.odb).
- BUILDINGSMODE_ASCII : Load buildings from ASCII WinProp file (.oda).

WINPROP_INDOOR_COVERAGE_MODE

- COVERAGEINDOOR_NONE : Indoor coverage not required.
- COVERAGEINDOOR_CONSTANT : Indoor coverage computed with constant level model.
- COVERAGEINDOOR_EXPONENTIAL : Indoor coverage computed with exponential decrease model.
- COVERAGEINDOOR_USERDEFINED : Indoor coverage computed with variable decrease model.
- COVERAGEINDOOR_ONROOF : Coverage computed on roof.

WINPROP_OUTDOOR_TOPO_MODE

- TOPOMODE_NONE : No topography considered. See WinProp_ParaMain::TopographyMode where this is used.
- TOPOMODE_RAM : Load topography from RAM. See WinProp_ParaMain::TopographyMode where this is used.
- TOPOMODE_FILE : Load topography from the building database. See WinProp_ParaMain::TopographyMode where this is used.
- TOPOMODE_SEPARATE : load topography from a separate file. See WinProp_ParaMain::TopographyMode where this is used.

WINPROP_VEGETATION_MODE

- VEGETATION_NONE : No vegetation considered. See WinProp_ParaMain::VegetationMode where this is used.
- VEGETATION_PIXEL : Load from a .veg file. See WinProp_ParaMain::VegetationMode where this is used.
- VEGETATION_SEPARATE : Load from a separate .odb file. See WinProp_ParaMain::VegetationMode where this is used.
- VEGETATION_BUILDING : Load from building databse. See WinProp_ParaMain::VegetationMode where this is used.
- VEGETATION_CLUTTER : Load from clutter database. See WinProp_ParaMain::VegetationMode where this is used.

WINPROP_CLUTTER_MODE

- CLUTTERMODE_NONE : No clutter considered. See WinProp_ParaMain::ClutterMode where this is used.
- CLUTTERMODE_RAM : Load clutter from RAM. See WinProp_ParaMain::ClutterMode where this is used.
- CLUTTERMODE_FILE : Load clutter from a separate file. See WinProp_ParaMain::ClutterMode where this is used.

WINPROP_URBAN_IRT_MODE

- PP_MODE_IRT_3D : Setting for 3D IRT. See WinProp_PreProUrban::Mode where this is used.
- PP_MODE_IRT_2x2D_IRT : Setting for 2D horizontal IRT and 2D vertical IRT. See WinProp_PreProUrban::Mode where this is used.

- PP_MODE_IRT_2x2D_KE : Setting for 2D horizontal IRT and vertical Knife Edge model. See WinProp_PreProUrban::Mode where this is used.

WINPROP_URBAN_IRT_ACC_ZONE

- PP_MODE_IRT_ZONE_OFF : No spheric zone considered. See WinProp_PreProUrban::SphericMode where this is used.
- PP_MODE_IRT_ZONE_CONSTANT : Constant spheric zone size. See WinProp_PreProUrban::SphericMode where this is used.
- PP_MODE_IRT_ZONE_ADAPTIVE : Adaptive spheric zone size. See WinProp_PreProUrban::SphericMode where this is used.

WINPROP_KE_MODEL

- KE_OFF : Not considered. See WinProp_ParaRural::KnifeEdge where this is used.
- KE_DEYGOUT : Considered according to the approach from Deygout. See WinProp_ParaRural::KnifeEdge where this is used.
- KE_EPSTEIN : Considered according to the approach from Epstein/Peterson. See WinProp_ParaRural::KnifeEdge where this is used.

WINPROP_HATA_SUB_MODEL

- HATA_SUB_CLUTTERTABLE : as defined in clutter table See WinProp_ParaRural::HataSubMode where this is used.
- HATA_SUB_DENSEURBAN : dense urban See WinProp_ParaRural::HataSubMode where this is used.
- HATA_SUB_MEDIUMURBAN : medium urban See WinProp_ParaRural::HataSubMode where this is used.
- HATA_SUB_SUBURBAN : suburban See WinProp_ParaRural::HataSubMode where this is used.
- HATA_SUB_OPEN : open area See WinProp_ParaRural::HataSubMode where this is used.

The documentation was generated from the following file:

- `source/Interface/OutdoorPlugIn_Defines.h`

2.2.12 Callbacks for simulation progress

Detailed Description

Wave propagation interface callbacks.

Callback function pointers

typedef int(_STD_CALL * CALLBACK_PERCENTAGE) (int, const char *)
: Percentage progress.

typedef int(_STD_CALL * CALLBACK_MESSAGEINFO) (const char *)
: Callback message string.

typedef int(_STD_CALL * CALLBACK_ERROR) (const char *, int)
: Callback error information string.

typedef int(_STD_CALL * CALLBACK_TERMINATE) ()
: Callback to check for termination, returning a value unequal to zero from the callback requests the computation to stop.

The documentation was generated from the following file:

- `source/Interface/WP_Callback.h`

2.2.13 Defines and constants

Detailed Description

Wave propagation interface defines.

Definition of Legend palettes.

LEGEND_PALETTE_MAXIMUM_REGULAR *1536*
: Size of regular legend palette.

LEGEND_PALETTE_MAXIMUM_PROMAN *1280*
: Size of legend palette in ProMan.

Definition of the size of the antenna pattern loaded from RAM

WINPROP_SIZE_PATTERN_VERTICAL *180*
: Number of samples in the vertical cut of the pattern loaded from RAM (See the setting in WinProp_Pattern::Mode)

WINPROP_SIZE_PATTERN_HORIZONTAL *360*
: Number of samples in the horizontal cut of the pattern loaded from RAM (See the setting in WinProp_Pattern::Mode)

Enum

WINPROP_ANTENNA_MODE
: - ANTENNA_DISABLED : Antenna disabled.
- ANTENNA_ENABLED : Antenna enabled.
- ANTENNA_ENABLED_FIXED : Antenna enabled at a fixed site.

WINPROP_REPEATER_MODE
: - WINPROP_REPEATER_MODE_OFF : Antenna repeater mode is OFF.
- WINPROP_REPEATER_MODE_RECONFIGURABLE : Antenna repeater is in reconfigurable mode.

- WINPROP_REPEATER_MODE_TRANSPARENT : Antenna repeater is in transparent mode.

WINPROP_ANTENNA_POLARIZATION_MODE

- ANTENNA_POLARIZATION_MODE_UNKNOWN : Antenna polarisation unknown.
- ANTENNA_POLARIZATION_MODE_LINEAR_VERTICAL : Antenna polarisation linear vertical.
- ANTENNA_POLARIZATION_MODE_LINEAR_HORIZONTAL : Antenna polarisation linear horizontal.
- ANTENNA_POLARIZATION_MODE_LINEAR_PLUS_45 : Antenna polarisation linear +45 deg.
- ANTENNA_POLARIZATION_MODE_LINEAR_MINUS_45 : Antenna polarisation linear -45 deg.
- ANTENNA_POLARIZATION_MODE_LINEAR_ARBITRARY : Antenna polarisation linear user defined.
- ANTENNA_POLARIZATION_MODE_RHCP : Antenna polarisation right hand circular.
- ANTENNA_POLARIZATION_MODE_LHCP : Antenna polarisation left hand circular.
- ANTENNA_POLARIZATION_MODE_X_POL : Antenna cross polarisation.

WINPROP_SCENARIO

- WINPROP_SCENARIO_URBAN : Urban scenario. This is used in WinProp_Scenario::Scenario.
- WINPROP_SCENARIO_INDOOR : Indoor scenario. This is used in WinProp_Scenario::Scenario.
- WINPROP_SCENARIO_RURAL : Rural scenario. This is used in WinProp_Scenario::Scenario.

WINPROP_MODEL_INDOOR

- WINPROP_MODEL_FREESPACE : Free space model. See Model_FREESPACE.
- WINPROP_MODEL_COST231 : COST 231 model. See Model_COST.
- WINPROP_MODEL_DPM : Dominant path model. See Model_DPM.
- WINPROP_MODEL_IRT : Intelligent ray tracing model. See Model_RAYTRACING.
- WINPROP_MODEL_SRT : Standard ray tracing model. See Model_RAYTRACING.
- WINPROP_MODEL_VRT : Vertical ray tracing model. See Model_VRT.
- WINPROP_MODEL_SBR : Shooting & bouncing rays model. See Model_RAYTRACING.
- WINPROP_MODEL_MOTLEYKEENAN : Motley-Keenan model. See Model_MOTLEYKEENAN.

WINPROP_RESULT_TYPE

- WINPROP_RESULT_UNKNOWN : Unknown result type.
- PROP_RESULT_POWER : Received power.
- PROP_RESULT_FIELD : Field strength.
- PROP_RESULT_PATHLOSS : Path loss.

- PROP_RESULT_DELAY : Delay.
- PROP_RESULT_PATHTYPE : Type of path, see PROP_RESULT_PATHTYPES_T.
- PROP_RESULT_DELAY_SPREAD : Delay spread.
- PROP_RESULT_ELEVATION_SPREAD_BS : Elevation spread BS.
- PROP_RESULT_ELEVATION_SPREAD_MS : Elevation spread MS.
- PROP_RESULT_ELEVATION_MEAN_BS : Elevation mean BS.
- PROP_RESULT_ELEVATION_MEAN_MS : Elevation mean MS.
- PROP_RESULT_ANGULAR_SPREAD_BS : Angular spread BS.
- PROP_RESULT_ANGULAR_SPREAD_MS : Angular spread MS.
- PROP_RESULT_ANGULAR_MEAN_BS : Angular mean BS.
- PROP_RESULT_ANGULAR_MEAN_MS : Angular mean MS.
- PROP_RESULT_MIN_INTERACTIONS : Minimal number of interactions.
- NET_RESULT_BEST_SERVER : Best server results.
- NET_RESULT_CELL_AREA : Cell area results.
- NET_RESULT_REC_PROB : Receiver probability.
- NET_RESULT_MAX_DATARATE : Maximum data rate.
- NET_RESULT_MAX_REC_POWER : Maximum receiver power.
- NET_RESULT_SNIR : SNIR.
- NET_RESULT_NR_REC_CARRIERS : Number of carriers.
- NET_RESULT_MIN_REQ_MS_TX_PWR : Minimum required TX power at base station.
- NET_RESULT_MIN_REQ_BS_TX_PWR : Minimum required TX power at mobile station.
- NET_RESULT_HANDOVER : Handover.
- NET_RESULT_CDMA_EC_OVER_NT : Ec / Nt.
- NET_RESULT_LTE_RSRP : RSRP for LTE.
- NET_RESULT_LTE_RSRQ : RSRQ for LTE.
- NET_RESULT_LTE_RSSI : RSSI for LTE.
- NET_RESULT_MAX_THROUGHPUT : Maximum throughput.
- NET_RESULT_MIMO_NUMBER_STREAMS : Number of MIMO streams.
- NET_RESULT_MIMO_NUMBER_STREAMS_UL : Number of MIMO streams (uplink).
- NET_RESULT_REC_PROB_UL : Receiver probability (uplink).
- NET_RESULT_MAX_DATARATE_UL : Maximum data rate (uplink).
- NET_RESULT_MAX_REC_POWER_UL : Maximum receiver power (uplink).
- NET_RESULT_SNIR_UL : SNIR (uplink).
- NET_RESULT_MAX_THROUGHPUT_UL : Maximum throughput (uplink).
- NET_RESULT_CDMA_EC_OVER_NT_UL : Ec / Nt (uplink).
- NET_RESULT_CDMA_MAX_EC : Maximum Ec.
- NET_RESULT_CDMA_MAX_EC_UL : Maximum Ec (uplink).
- NET_RESULT_NR_REC_TRX : Number of transmitters.
- NET_RESULT_COVERAGE_PROB : Coverage probability.

- NET_RESULT_THROUGHPUT : Throughput per transmission mode.
- NET_RESULT_THROUGHPUT_UL : Throughput per transmission mode (uplink).
- NET_RESULT_DATARATE : Data rate per transmission mode.
- NET_RESULT_DATARATE_UL : Data rate per transmission mode (uplink).
- NET_RESULT_TOTAL_REC : Total received signal (signal + noise + interference).
- NET_RESULT_INTERFERENCE_NOISE_REC : Interference level (noise + interference).
- NET_RESULT_INTERFERENCE_NOISE_REC_UL : Interference level (noise + interference) (uplink).
- NET_RESULT_REC_POWER : Received power per transmission mode.
- NET_RESULT_REC_POWER_UL : Received power per transmission mode (uplink).
- NET_RESULT_NR_PARALLEL_USERS : Number of parallel users per transmission mode.
- NET_RESULT_NR_PARALLEL_USERS_UL : Number of parallel users per transmission mode (uplink).
- NET_RESULT_SIGNAL_AREA : Signal area.
- NET_RESULT_EBN0 : Max. achievable Eb/N0.
- NET_RESULT_EBN0_UL : Max. achievable Eb/N0 (uplink).
- NET_RESULT_NR_REC_SITE : Number of Sites.
- NET_RESULT_SITE_AREA : Site area.
- NET_RESULT_POWER_SUPERPOSITION : Power superposition.
- NET_RESULT_NEIGHBOUR_CELLS : Neighbor cell list.
- NET_RESULT_TRAFFIC_OFFERED : Offered traffic.
- NET_RESULT_TRAFFIC_SERVED : Served traffic.
- NET_RESULT_TRAFFIC_BLOCKED : Blocked traffic.
- NET_RESULT_TRAFFIC_STATE : Traffic state.
- NET_RESULT_TRAFFIC_NR_USERS_GEN : Number of generated users.
- NET_RESULT_EMC_FIELD : EMC/EMF field.
- NET_RESULT_EXPOSURE : EMF total exposure.
- NET_RESULT_EXPOSURE_ZONE : EMF exposure zones.
- NET_RESULT_OFDM_POWER : Received power for OFDM broadcasting air interface.
- NET_RESULT_OFDM_INTERFERENCE : SNIR for OFDM broadcasting air interface.
- NET_RESULT_BEAM_ASSIGNMENT_CTRL : Antenna Beam Assignment Ctrl
- NET_RESULT_BEAM_ASSIGNMENT_DATA : Antenna Beam Assignment Data

PROP_RESULT_PATHTYPES_T

- PROP_RESULT_PATHTYPE_UNKNOWN : Unknown path type.
- PROP_RESULT_PATHTYPE_LOS : Line of sight (LOS) path.
- PROP_RESULT_PATHTYPE_OLOS : Obstructed line of sight (OLOS) path.
- PROP_RESULT_PATHTYPE_NLOS : Non-line of sight (NLOS) path.
- PROP_RESULT_PATHTYPE_VLOS : Line of sight (NLOS) path, which travels through vegetation (urban) or furniture (indoor).

- PROP_RESULT_PATHTYPE_VNLOS : Non-line of sight (NLOS) path, which travels through vegetation (urban) or furniture (indoor).

WINPROP_POWER_MODE

- WINPROP_POWER_OUTPUT : Output power amplifier.
- WINPROP_POWER_EIRP : EIRP.
- WINPROP_POWER_ERP : ERP.

WINPROP_ANTENNA_TYPE

- WINPROP_ANTENNA_ORDINARY : Ordinary antenna. This is used in WinProp_Antenna::Type.
- WINPROP_ANTENNA_RADIATING_CABLE : Radiating cable. This is used in WinProp_Antenna::Type.
- WINPROP_ANTENNA_SATELLITE : Satellite (Geostationary and Non-stationary). This is used in WinProp_Antenna::Type.
- WINPROP_ANTENNA_GPS_SATELLITE : GPS Satellite (Almanac and TLE). This is used in WinProp_Antenna::Type.
- WINPROP_ANTENNA_EXTERNAL : External leakage interference. This is used in WinProp_Antenna::Type.

WINPROP_INTERACTION_MODEL

- WINPROP_INTERACTION_MODEL_EMPIRICAL : Empirical losses for reflection, transmission and diffraction.
- WINPROP_INTERACTION_MODEL_GTDUTD : Fresnel coefficients for reflection and transmission, and GTD/UTD for diffraction.

WINPROP_RAD_CABLE_MODEL

- WINPROP_MODEL_ID_CABLE_SDM : Shortest distance model.
- WINPROP_MODEL_ID_CABLE_STL : Smallest transmission loss model.
- WINPROP_MODEL_ID_CABLE_SPL : Smallest path loss model.
- WINPROP_MODEL_ID_CABLE_DPM : Dominant path model.

WINPROP_BREAKPOINT_MODE

- BREAKPOINT_DISABLE : Disable breakpoint effect. See WinProp_Additional::BreakpointMode where this is used. Relevant for Standard ray tracing model.
- BREAKPOINT_DEFAULT : Default determination of breakpoint. See WinProp_ParaMain::BreakpointMode and WinProp_Additional::BreakpointMode where this is used.
- BREAKPOINT_FIXED : Breakpoint with fixed distance. See WinProp_ParaMain::BreakpointMode and WinProp_Additional::BreakpointMode where this is used.

- BREAKPOINT_MEAN_BUILDING : Breakpoint computed with transmitter height relative to mean building height in a radius of 1000m around it. See WinProp_ParaMain::BreakpointMode where this is used.

WINPROP_PRED_PLANE_TYPE

- WINPROP_PRED_PLANE_TYPE_AREA_ABSOLUTE : Result corresponds to area prediction at absolute height.
- WINPROP_PRED_PLANE_TYPE_AREA_RELATIVE : Result corresponds to area prediction at relative height.
- WINPROP_PRED_PLANE_TYPE_PLANE : Result corresponds to a user specified prediction plane (either via parameters or from the database)
- WINPROP_PRED_PLANE_TYPE_SURFACE : Result corresponds to a predicted surface of a database object.
- WINPROP_PRED_PLANE_TYPE_HORIZONTAL : Result corresponds to a horizontal prediction, in urban scenarios to CNP layers.
- WINPROP_PRED_PLANE_TYPE_VERTICAL : Result corresponds to a vertical prediction.
- WINPROP_PRED_PLANE_TYPE_USER_POINT : Result corresponds to a user point.

WINPROP_PATTERN_MODE

- PATTERN_MODE_2X2D : Total pattern given by 2 2-D patterns (vertical and horizontal polarisations). See WinProp_Pattern::Mode.
- PATTERN_MODE_3D : Total pattern given by a 3-D pattern. See WinProp_Pattern::Mode.
- PATTERN_MODE_FILE : Total pattern read from a file. See WinProp_Pattern::Mode.
- PATTERN_MODE_ANTENNA_ARRAY : Total pattern generated by an antenna array WinProp_Antenna_Array. See WinProp_Pattern::Mode.

WINPROP_PATTERN_RESOLUTION

- WINPROP_PATTERN_RESOLUTION_1DEG : 1.00 degrees pattern resolution
- WINPROP_PATTERN_RESOLUTION_05DEG : 0.50 degrees pattern resolution
- WINPROP_PATTERN_RESOLUTION_033DEG : 0.33 degrees pattern resolution
- WINPROP_PATTERN_RESOLUTION_025DEG : 0.25 degrees pattern resolution
- WINPROP_PATTERN_RESOLUTION_020DEG : 0.20 degrees pattern resolution
- WINPROP_PATTERN_RESOLUTION_01DEG : 0.10 degrees pattern resolution

TOTAL_POWER_MODE

- INCOHERENT : Incoherent superposition mode.
- COHERENT : Coherent superposition mode.

Other defines

typedef std::vector<WinProp_Antenna_Area> WinPropAntennaList_t
typedef for a vector of WinProp_Antenna_Area.

Other defines

WALL_STANDARD *0*

WALL_VEGETATION *2*

WALL_COURTYARD *3*

The documentation was generated from the following file:

- `source/Public/Interface/WallTypes.h`

2.2.14 Handling Map and Vector Data

Functions in this section are used to handle map and vector data.

This chapter contains the following:

1. Functions for Buildings
2. Functions for Clutter
3. Functions for walls
4. Defines and constants
5. Functions for Materials
6. Functions for Topography

2.2.15 Functions for Buildings

Function List

InterfaceLoadBuildingsAircomSingleFile
: Load buildings from a single AIRCOM file.

InterfaceLoadBuildingsASCII
: Reads building data from a *.oda ASCII file.

InterfaceBuildingInit
: Initialise interface buildings.

InterfaceBuildingAllocat)
: Allocate Memory to Building data. URBAN_BUILDING the allocated building.

InterfaceBuildingCornersAllocate
: Allocate Corners of Buildings.

InterfaceBuildingCornersFree
: Free building corners

InterfaceBuildingFree
: Free building structure

InterfaceBuildingsInit
: Initialise building structure of DBInterface.dll.

InterfaceBuildingsCopy
: Copy building data.

InterfaceBuildingsAdd
: Add new buildings

InterfaceBuildingsFree
: Free building data.

InterfaceBuildingsSort
: Sort building structure according to building height.

InterfaceBuildingsWriteASCII
: Write building data to an ASCII database.

Function Details

void ***InterfaceLoadBuildingsAircomSingleFile****(*URBAN_BUILDINGS * *Buildings, const char * FilenameBuildings, const char * FilenameAttributes)*

Description

Load buildings from a single AIRCOM file.

Parameters

URBAN_BUILDINGS * *Buildings*
: Building data.

*const char * FilenameBuildings*
: Filename of buildings file.

*const char * FilenameAttributes*
: Filename of attributes file.

Returns None

int ***InterfaceLoadBuildingsASCII****(*URBAN_BUILDINGS * *Buildings, const char * Filename)*

Description

Reads building data from a *.oda ASCII file.

Parameters

URBAN_BUILDINGS * *Buildings*
: Building data.

*const char * Filename*
: Filename of buildings file.

Returns An int 0 in case of success, != 0 otherwise.

void ***InterfaceBuildingInit****(*URBAN_BUILDING * *building)*

Description

Initialise interface buildings.

Parameters

URBAN_BUILDING * *building*
Building data.

Returns None

URBAN_BUILDING ***InterfaceBuildingAllocate****(INTERFACE_API URBAN_BUILDING * InterfaceBuildingAllocat)*

Description

Allocate Memory to Building data. URBAN_BUILDING the allocated building.

Parameters

Returns None

INTERFACE_CORNER ***InterfaceBuildingCornersAllocate****(unsigned int NbrCorners)*

Description

Allocate Corners of Buildings.

Parameters

unsigned int NbrCorners
Number of corners.

Returns None

int ***InterfaceBuildingCornersFree****(*INTERFACE_CORNER ** *Ptr)*

Description

Free building corners

Parameters

INTERFACE_CORNER ** *Ptr*
Corners to be freed.

Returns 0.

void ***InterfaceBuildingFree****(*URBAN_BUILDING * *pBuilding)*

Description

Free building structure

Parameters

URBAN_BUILDING * *pBuilding*
Urban building structure.

Returns None

void ***InterfaceBuildingsInit****(*URBAN_BUILDINGS * *buildings)*

Description

Initialise building structure of DBInterface.dll.

Parameters

URBAN_BUILDINGS * *buildings*
Building data.

Returns None

int ***InterfaceBuildingsCopy****(*URBAN_BUILDINGS * *copy, const* URBAN_BUILDINGS * *source)*

Description

Copy building data.

Parameters

URBAN_BUILDINGS * *copy*
Copy of building data.

const URBAN_BUILDINGS * *source*
Source building data.

Returns An int.

URBAN_BUILDING ***InterfaceBuildingsAdd****(unsigned int nbrCorners, bool bInsertTail, bool bMaterial,* URBAN_BUILDINGS * *pBuildingList)*

Description

Add new buildings

Parameters

unsigned int nbrCorners
Number of corners of the new building.

bool bInsertTail
Insert at the tail of the list

bool bMaterial
Allocate sub structure for material data.

URBAN_BUILDINGS * *pBuildingList*
List with building data to add new building.

Returns A pointer to an

void ***InterfaceBuildingsFree****(*URBAN_BUILDINGS * *buildings)*

Description

Free building data.

Parameters

URBAN_BUILDINGS * *buildings*
Building data.

Returns None

int ***InterfaceBuildingsSort****(*URBAN_BUILDINGS * *buildings, int criteria)*

Description

Sort building structure according to building height.

Parameters

URBAN_BUILDINGS * *buildings*
Building data.

int criteria
Criteria for sorting. Specified by SORT_BUILDING_HEIGHT_INC or SORT_BUILDING_HEIGHT_DEC

Returns An int.

int ***InterfaceBuildingsWriteASCII****(*URBAN_BUILDINGS * *buildings, const char * databaseName)*

Description

Write building data to an ASCII database.

Parameters

URBAN_BUILDINGS * *buildings*
Building data.

*const char * databaseName*
Name of the database.

Returns An int.

The documentation was generated from the following file:

- `source/Public/Interface/Buildings.h`

2.2.16 Functions for Clutter

Function List

InterfaceFrequencyLossInit
Initialise frequency loss structure.

InterfaceFrequencyLossAlloc
Memory allocation for frequency loss structure

InterfaceFrequencyLossCopy
Copy frequency loss structure.

InterfaceFrequencyLossFree
Free allocated memory for frequency loss structure.

InterfaceClutterClassInit
Initialise clutter class structure

InterfaceClutterClassAlloc
Memory allocation for clutter class structure.

InterfaceClutterClassCopy
: Copy clutter class structure.

InterfaceClutterClassFree
: Free allocated memory for clutter class structure.

InterfaceClutterInit
: Initialise clutter data structure.

InterfaceClutterAlloc
: Memory allocation for clutter structure Interface.

InterfaceClutterCopy
: Copy clutter data structure.

InterfaceClutterFree
: Free clutter data structure.

InterfaceClutterWriteASCII
: Write clutter database to ASCII *.asc file in grid format.

InterfaceClutterPropertiesWriteASCII
: Write clutter properties to ASCII *.mct file format.

Function Details

void ***InterfaceFrequencyLossInit****(*FREQUENCY_LOSS * *frequencyLoss)*

Description

Initialise frequency loss structure.

Parameters

FREQUENCY_LOSS * *frequencyLoss*
: Frequency loss structure.

Returns None

int ***InterfaceFrequencyLossAlloc****(*FREQUENCY_LOSS * *frequencyLoss)*

Description

Memory allocation for frequency loss structure

Parameters

FREQUENCY_LOSS * *frequencyLoss*
: Frequency loss structure.

Returns An integer: 0 = data valid

int ***InterfaceFrequencyLossCopy****(*FREQUENCY_LOSS * *copy, const* FREQUENCY_LOSS * *source)*

Description

Copy frequency loss structure.

Parameters

FREQUENCY_LOSS * *copy*
Frequency loss structure (copy).

const FREQUENCY_LOSS * *source*
Frequency loss structure to be copied.

Returns An integer: 0 = data valid

void ***InterfaceFrequencyLossFree****(*FREQUENCY_LOSS * *frequencyLoss)*

Description

Free allocated memory for frequency loss structure.

Parameters

FREQUENCY_LOSS * *frequencyLoss*
Frequency loss structure.

Returns None

void ***InterfaceClutterClassInit****(*CLUTTER_CLASS * *clutterClass)*

Description

Initialise clutter class structure

Parameters

CLUTTER_CLASS * *clutterClass*
Clutter class structure.

Returns None

int ***InterfaceClutterClassAlloc****(int nbrAttenuations,* CLUTTER_CLASS * *clutterClass)*

Description

Memory allocation for clutter class structure.

Parameters

int nbrAttenuations
Number of attenuations.

CLUTTER_CLASS * *clutterClass*
Clutter class structure.

Returns An integer: 0 = data valid

int ***InterfaceClutterClassCopy****(*CLUTTER_CLASS * *copy, const* CLUTTER_CLASS * *source)*

Description

Copy clutter class structure.

Parameters

CLUTTER_CLASS * *copy*
Clutter class structure (copy).

const CLUTTER_CLASS * *source*
Clutter class structure to be copied.

Returns An integer: 0 = data valid

void ***InterfaceClutterClassFree****(*CLUTTER_CLASS * *clutterClass)*

Description

Free allocated memory for clutter class structure.

Parameters

CLUTTER_CLASS * *clutterClass*
Clutter class structure.

Returns None

void ***InterfaceClutterInit****(*CLUTTER * *clutter)*

Description

Initialise clutter data structure.

Parameters

CLUTTER * *clutter*
data structure.

Returns None

int ***InterfaceClutterAlloc****(int nbrColumns, int nbrLines, int nbrClutterClasses,* CLUTTER * *clutter)*

Description

Memory allocation for clutter structure Interface.

Parameters

int nbrColumns
Number of columns.

int nbrLines
Number of lines.

int nbrClutterClasses
Number of clutter classes.

CLUTTER * *clutter*
Clutter data structure.

Returns An integer: 0 = data valid

int ***InterfaceClutterCopy****(*CLUTTER * *copy, const* CLUTTER * *source)*

Description

Copy clutter data structure.

Parameters

CLUTTER * *copy*
Copy of clutter data.

const CLUTTER * *source*
Source clutter data.

Returns An integer: 0 = data valid

void ***InterfaceClutterFree****(*CLUTTER * *clutter)*

Description

Free clutter data structure.

Parameters

CLUTTER * *clutter*
Clutter data structure.

Returns None

int ***InterfaceClutterWriteASCII****(*CLUTTER * *clutter, const char * databaseName)*

Description

Write clutter database to ASCII *.asc file in grid format.

Parameters

CLUTTER * *clutter*
Non-null, the clutter structure.

*const char * databaseName*
Filepath for output file.

Returns An int - 0 on success, !0 on failure.

int ***InterfaceClutterPropertiesWriteASCII****(*CLUTTER * *clutter, const char * databaseName)*

Description

Write clutter properties to ASCII *.mct file format.

Parameters

CLUTTER * *clutter*
Non-null, the clutter structure.

*const char * databaseName*
Filepath for output file.

Returns An int - 0 on success, !0 on failure.

The documentation was generated from the following file:

- `source/Public/Interface/Clutter.h`

2.2.17 Functions for walls

Function List

InterfaceWallInit
: Initialise one wall.

InterfaceWallsInit
: Initialise a list of walls

InterfaceWallsAlloc
: Memory allocation of walls describing a scenario.

InterfaceWallCornersInit
: Initialise wall corners

InterfaceWallCornersAlloc
: Allocation of memory for corners describing a wall.

InterfaceWallCornersFree
: Free memory of corners.

InterfaceWallsFree
: Free memory allocated to a list of walls.

Function Details

int ***InterfaceWallInit****(*INDOOR_WALL * *Wall)*

Description

Initialise one wall.

Parameters

INDOOR_WALL * *Wall*
: Wall to initialise.

Returns 0.

int ***InterfaceWallsInit****(*INDOOR_WALLS * *Walls)*

Description

Initialise a list of walls

Parameters

INDOOR_WALLS * *Walls*
: Walls to initialise.

Returns 0.

int ***InterfaceWallsAlloc****(int NrWalls,* INDOOR_WALLS * *Walls)*

Description

Memory allocation of walls describing a scenario.

Parameters

int NrWalls
The number of walls.

INDOOR_WALLS * *Walls*
List of walls.

Returns 0.

int ***InterfaceWallCornersInit****(*COORDPOINT * *Point)*

Description

Initialise wall corners

Parameters

COORDPOINT * *Point*
Corner point.

Returns 0.

int ***InterfaceWallCornersAlloc****(int NrCorners,* INDOOR_WALL * *Wall)*

Description

Allocation of memory for corners describing a wall.

Parameters

int NrCorners
The number corners.

INDOOR_WALL * *Wall*
The wall.

Returns 0.

int ***InterfaceWallCornersFree****(*INDOOR_WALL * *Wall)*

Description

Free memory of corners.

Parameters

INDOOR_WALL * *Wall*
The wall to be freed.

Returns 0.

int ***InterfaceWallsFree****(*INDOOR_WALLS * *Walls)*

Description

Free memory allocated to a list of walls.

Parameters

INDOOR_WALLS * *Walls*
Walls to be freed.

Returns 0.

The documentation was generated from the following file:

- `source/Public/Interface/IndoorWalls.h`

2.2.18 Defines and constants

Detailed Description

building databases

BUILDING_TYPE_AIRCOM *1*
: AIRCOM buildingdatabase format.

BUILDING_TYPE_WINPROP *2*
: WinProp building database format.

topo databases

TOPO_TYPE_AIRCOM *1*
: AIRCOM topography database format.

TOPO_TYPE_WINPROP *2*
: WinProp topography database format.

clutter databases

CLUTTER_TYPE_AIRCOM *1*
: AIRCOM clutter database format.

CLUTTER_CLASS_NAME_LENGTH *400*
: WinProp clutter database format.

Sort building height in database.

SORT_BUILDING_HEIGHT_INC *0*
: sort buildings according to increasing height

SORT_BUILDING_HEIGHT_DEC *1*
: sort buildings according to decreasing height

user info

PROGRESS_TEXT_1 *"Reading databases..."*
: Database processing progress text.

PROGRESS_TEXT_2 *"of vector database read"*
: Database processing progress text.

PROGRESS_TEXT_3 *"of topography database read."*
Database processing progress text.

PROGRESS_TEXT_4 *"of clutter database read"*
Database processing progress text.

Error codes

ERROR_1 *"No memory available!"*
Error when processing building database.

ERROR_2 *"Not possible to open database file!"*
Error when processing building database.

ERROR_3 *"Error while reading building vector data!"*
Error when processing building database.

ERROR_4 *"No building data available!"*
Error when processing building database.

Other defines

MATERIAL_INDEX_NOT_DEFINED *-1*
Material index not defined.

The documentation was generated from the following file:

- `source/Public/Interface/IndoorWalls.h`

2.2.19 Functions for Materials

Function List

InterfaceMaterialInit
Initialise interface material.

InterfaceMaterialBandInit
Initialise frequency dependant material properties.

InterfaceMaterialBandsAlloc
Allocation of materials.

InterfaceMaterialsAlloc
Allocation of memory for material bands of a material.

InterfaceMaterialBandsFree
Free memory of all material bands of a material.

InterfaceMaterialsFree
Free memory allocated to materials.

InterfaceMaterialsInit
Initialize materials.

InterfaceMaterialInit
: Initialise material structure

InterfaceMaterialCopy
: Initialise material structure

InterfaceMaterialFree
: Free material structure

Function Details

int ***InterfaceMaterialInit****(*MATERIAL * *Material)*

Description

Initialise interface material.

Parameters

MATERIAL * *Material*
: The material.

Returns 0.

int ***InterfaceMaterialBandInit****(*MATERIALBAND * *Band)*

Description

Initialise frequency dependant material properties.

Parameters

MATERIALBAND * *Band*
: Material properties to initialise.

Returns 0.

int ***InterfaceMaterialBandsAlloc****(int NrBands,* MATERIAL * *Material)*

Description

Allocation of materials.

Parameters

int NrBands
: Number of materials.

MATERIAL * *Material*
: Material structure of type MATERIALS.

Returns 0.

int ***InterfaceMaterialsAlloc****(int NrMaterials,* MATERIALS * *Materials)*

Description

Allocation of memory for material bands of a material.

Parameters

int NrMaterials
Number of bands.

MATERIALS * *Materials*
Material structure of type MATERIAL.

Returns 0.

int ***InterfaceMaterialBandsFree****(*MATERIAL * *Material)*

Description

Free memory of all material bands of a material.

Parameters

MATERIAL * *Material*
All material band of MATERIAL will be freed.

Returns 0.

int ***InterfaceMaterialsFree****(*MATERIALS * *Materials)*

Description

Free memory allocated to materials.

Parameters

MATERIALS * *Materials*
material to be freed.

Returns 0.

int ***InterfaceMaterialsInit****(*MATERIALS * *Materials)*

Description

Initialize materials.

Parameters

MATERIALS * *Materials*
Material to be initialised.

Returns 0.

void ***InterfaceMaterialInit****(*INTERFACE_MATERIAL * *material)*

Description

Initialise material structure

Parameters

INTERFACE_MATERIAL * *material*
The material.

Returns None

int ***InterfaceMaterialCopy****(*INTERFACE_MATERIAL * *source,* INTERFACE_MATERIAL * *copy)*

Description

Initialise material structure

Parameters

INTERFACE_MATERIAL * *source*
Source material data.

INTERFACE_MATERIAL * *copy*
Copy of material data.

Returns An integer. 0 = data valid

void ***InterfaceMaterialFree****(*INTERFACE_MATERIAL * *material)*

Description

Free material structure

Parameters

INTERFACE_MATERIAL * *material*
the material struct to be freed.

Returns None

The documentation was generated from the following file:

- `source/Public/Interface/InterfaceMaterial.h`

2.2.20 Functions for Topography

Function List

InterfaceTopoInit
Initialise topography structure of Interface

InterfaceTopoAlloc
Memory allocation for topography structure of Interface

InterfaceTopoCopy
Copy topography structure of Interface

InterfaceTopoFree
Free topography structure of Interface

InterfaceTopoWriteASCII
Write topography database to ASCII *.asc file in grid format.

InterfaceTopoWriteASCIILine
Write topography database to ASCII *.asc file in line format.

Function Details

void ***InterfaceTopoInit****(*TOPOGRAPHY * *topo)*

Description

Initialise topography structure of Interface

Parameters

TOPOGRAPHY * *topo*
Topography data.

Returns None

int ***InterfaceTopoAlloc****(int nbrColumns, int nbrLines,* TOPOGRAPHY * *topo)*

Description

Memory allocation for topography structure of Interface

Parameters

int nbrColumns
Number of columns.

int nbrLines
Number of lines.

TOPOGRAPHY * *topo*
Topography data structure.

Returns An integer: 0 = success, failure otherwise.

int ***InterfaceTopoCopy****(*TOPOGRAPHY * *copy, const* TOPOGRAPHY * *source)*

Description

Copy topography structure of Interface

Parameters

TOPOGRAPHY * *copy*
Copy of topography data.

const TOPOGRAPHY * *source*
Source topography data.

Returns An integer: 0 = success, failure otherwise.

void ***InterfaceTopoFree****(*TOPOGRAPHY * *topo)*

Description

Free topography structure of Interface

Parameters

TOPOGRAPHY * *topo*
Topography data to be freed.

Returns None

int ***InterfaceTopoWriteASCII****(*TOPOGRAPHY * *topo, const char * databaseName)*

Description

Write topography database to ASCII *.asc file in grid format.

Parameters

TOPOGRAPHY * *topo*
: Non-null, the topography structure.

*const char * databaseName*
: Filepath for output file.

Returns An int - 0 on success, !0 on failure.

int ***InterfaceTopoWriteASCIILine****(*TOPOGRAPHY * *topo, const char * databaseName)*

Description

Write topography database to ASCII *.asc file in line format.

Parameters

TOPOGRAPHY * *topo*
: Non-null, the topography structure.

*const char * databaseName*
: Filepath for output file.

Returns An int - 0 on success, !0 on failure.

The documentation was generated from the following file:

- `source/Public/Interface/Topo.h`

2.3 Propagation results

This chapter contains the following:

1. Network planning results

2.3.1 Network planning results

2.4 Mobile Station Postprocessing

Postprocessing of results at a mobile station.

This chapter contains the following:

1. Main Class
2. Defines and constants
3. Initialisation Functions

2.4.1 WinProp_SuperposeMS

Function List

WinProp_SuperposeMS::WinProp_SuperposeMS
: Default constructor

WinProp_SuperposeMS::~WinProp_SuperposeMS
: Destructor

WinProp_SuperposeMS::setArray
: Sets an antenna array.

WinProp_SuperposeMS::setArrayElement
: Sets antenna array element

WinProp_SuperposeMS::setPropagationArea
: Sets propagation input for the specified transmitting antenna. In case an array of type WINPROP_MS_ARRAY_INDIVIDUAL has been specified, a propagation result per array element is required. This method switches the computation mode to area wide computation, any other input propagation is removed.

WinProp_SuperposeMS::setPropagationPlanes
: Sets propagation input for the specified transmitting antenna. In case an array of type WINPROP_MS_ARRAY_INDIVIDUAL has been specified, a propagation result per array element is required. This method switches the computation mode to the computation of arbitrary planes, any other input propagation is removed.

WinProp_SuperposeMS::setPropagationPoints
: Sets propagation input for the specified transmitting antenna. In case an array of type WINPROP_MS_ARRAY_INDIVIDUAL has been specified, a propagation result per array element is required. This method switches the computation mode to the computation of individual points, any other input propagation is removed.

WinProp_SuperposeMS::setPropagationTrajectories
: Sets propagation input for the specified transmitting antenna. In case an array of type WINPROP_MS_ARRAY_INDIVIDUAL has been specified, a propagation result per array element is required. This method switches the computation mode to the computation of trajectories, any other input propagation is removed.

WinProp_SuperposeMS::compute
: Computes the superposition using the specified arrays and elements, based on the set propagation inputs.

WinProp_SuperposeMS::getResultArea
: Gets the results of an area wide computation.

WinProp_SuperposeMS::getResultPlane
: Gets the results of a computation on arbitrary planes.

WinProp_SuperposeMS::getResultPoints
: Gets the results of a point based computation.

WinProp_SuperposeMS::getObject

Function Details

WinProp_SuperposeMS()

Description

Default constructor

Parameters

Returns None

~WinProp_SuperposeMS()

Description

Destructor

Parameters

Returns None

int ***setArray****(bool Rx, WINPROP_MS_ARRAY_TYPE arrayType, int nrElements, float azimuth, float spacing, float radius, bool considerAntennaCoupling)*

Description

Sets an antenna array.

Parameters

bool Rx
: True to receiver array, false for transmitter array.

WINPROP_MS_ARRAY_TYPE *arrayType*
: Type of the array.

int nrElements
: The number of array elements.

float azimuth
: The array azimuth in degree (values between [0, 360]).

float spacing
: The array spacing in horizontal plane (k*Lambda), only relevant in case of arrayType set to WINPROP_MS_ARRAY_LINEAR.

float radius
: The array radius of circular array in meter, only relevant in case of arrayType set to WINPROP_MS_ARRAY_CIRCULAR.

bool considerAntennaCoupling
: True to consider antenna coupling.

Returns An integer: 0 = success, failure otherwise.

int ***setArrayElement****(bool Rx, int index, const* WinProp_Pattern * *pattern, const* WinProp_Pattern * *patternPolHoriz, float azimuth, float tilt,* WINPROP_MS_POLARIZATION_TYPE *polarization, const* COORDPOINT & *offset)*

Description

Sets antenna array element

Parameters

bool Rx
: True to receiver element, false for transmitter element.

int index
: Zero-based index of the element to set.

const WinProp_Pattern * *pattern*
: Specifies the antenna pattern of the element.

const WinProp_Pattern * *patternPolHoriz*
: Specifies the antenna pattern for horizontal polarization, needed for full polarimetric computations.

float azimuth
: The element azimuth in degree (values between [0, 360], with east=0deg over north=90deg orientation).

float tilt
: The element tilt in degree (values between [0, 180], a value of 90deg is equal to horizontal).

WINPROP_MS_POLARIZATION_TYPE *polarization*
: The polarization of the antenna, in case patternPolHoriz is set to nullptr.

const COORDPOINT & *offset*
: The offset of an individual receiving antenna element, only relevant for array of type WINPROP_MS_ARRAY_INDIVIDUAL.

Returns An integer: 0 = success, failure otherwise.

int ***setPropagationArea****(int trxIndex, const* WinProp_Antenna & *antenna, const* WinProp_RayMatrix & *rayMatrix)*

Description

Sets propagation input for the specified transmitting antenna. In case an array of type WINPROP_MS_ARRAY_INDIVIDUAL has been specified, a propagation result per array element is

required. This method switches the computation mode to area wide computation, any other input propagation is removed.

Parameters

int trxIndex
Zero-based index of the transmitting element.

const WinProp_Antenna *& antenna*
The antenna.

const WinProp_RayMatrix *& rayMatrix*
The ray matrix.

Returns An integer: 0 = success, failure otherwise.

int ***setPropagationPlanes****(int trxIndex, const* WinProp_Antenna *& antenna, const* WinProp_RayMatrixList *& rayMatrices)*

Description

Sets propagation input for the specified transmitting antenna. In case an array of type WINPROP_MS_ARRAY_INDIVIDUAL has been specified, a propagation result per array element is required. This method switches the computation mode to the computation of arbitrary planes, any other input propagation is removed.

Parameters

int trxIndex
Zero-based index of the transmitting element.

const WinProp_Antenna *& antenna*
The antenna.

const WinProp_RayMatrixList *& rayMatrices*
The ray matrices for the planes.

Returns An integer: 0 = success, failure otherwise.

int ***setPropagationPoints****(int trxIndex, const* WinProp_Antenna *& antenna, const* WinProp_ResultPointsList *& resPoints)*

Description

Sets propagation input for the specified transmitting antenna. In case an array of type WINPROP_MS_ARRAY_INDIVIDUAL has been specified, a propagation result per array element is required. This method switches the computation mode to the computation of individual points, any other input propagation is removed.

Parameters

int trxIndex
Zero-based index of the transmitting element.

const WinProp_Antenna *& antenna*
The antenna.

const WinProp_ResultPointsList *& resPoints*
The point results.

Returns An integer: 0 = success, failure otherwise.

int ***setPropagationTrajectories****(int trxIndex, const* WinProp_Antenna *& antenna, const* WinProp_Trajectory ** trajectories, const int nrTrajectories, const* WinProp_ResultTrajectoryList *& resultTrajectories)*

Description

Sets propagation input for the specified transmitting antenna. In case an array of type WINPROP_MS_ARRAY_INDIVIDUAL has been specified, a propagation result per array element is required. This method switches the computation mode to the computation of trajectories, any other input propagation is removed.

Parameters

int trxIndex
Zero-based index of the transmitting element.

const WinProp_Antenna *& antenna*
The antenna.

const WinProp_Trajectory ** trajectories*
The trajectories, an array of size nrTrajectories.

const int nrTrajectories
The number of trajectories.

const WinProp_ResultTrajectoryList *& resultTrajectories*
The trajectory results, needs to be of same size as trajectories.

Returns An integer: 0 = success, failure otherwise.

int ***compute****(const* WinProp_MS_Para *& msPara, const* WinProp_MS_AdditionalResults ** msAdditionalResult, const* WinProp_Callback ** callbacks)*

Description

Computes the superposition using the specified arrays and elements, based on the set propagation inputs.

Parameters

const WinProp_MS_Para *& msPara*
The parameters of the computation.

const WinProp_MS_AdditionalResults ** msAdditionalResult*
Optionally, specifies which additional results shall be written to files during the computation.

const WinProp_Callback ** callbacks*
If non-null, the callbacks to be used.

Returns An integer: 0 = success, failure otherwise.

int ***getResultArea****(int resultType,* WinProp_Result ** result,* WinProp_RayMatrix ** rayMatrix)*

Description

Gets the results of an area wide computation.

Parameters

int resultType
Type of the result.

WinProp_Result * *result*
If non-null, the result.

WinProp_RayMatrix * *rayMatrix*
If non-null, the ray matrix.

Returns An integer: 0 = success, failure otherwise.

int ***getResultPlane****(int resultType,* WinProp_ResultPlane * *result,* WinProp_RayMatrix * *rayMatrix)*

Description

Gets the results of a computation on arbitrary planes.

Parameters

int resultType
Type of the result.

WinProp_ResultPlane * *result*
If non-null, the result.

WinProp_RayMatrix * *rayMatrix*
If non-null, the ray matrix.

Returns An integer: 0 = success, failure otherwise.

int ***getResultPoints****(int resultType,* WinProp_ResultPointsList * *result)*

Description

Gets the results of a point based computation.

Parameters

int resultType
Type of the result.

WinProp_ResultPointsList * *result*
Non-null, the result.

Returns An integer: 0 = success, failure otherwise.

constWinProp_SuperposeMS_impl& ***getObject****()*

Description

Parameters

Returns None

The documentation was generated from the following file:

- `source/SuperposeMS/winprop_superposems.hpp`

2.4.2 Defines and constants

Detailed Description

Mobile station postprocessing structs and enums.

Enum

WINPROP_MS_POLARIZATION_TYPE

- WINPROP_MS_POLARIZATION_OMNI : An enum constant representing the omni polarization
- WINPROP_MS_POLARIZATION_VERTICAL : An enum constant representing the vertical polarization
- WINPROP_MS_POLARIZATION_HORIZONTAL : An enum constant representing the horizontal polarization
- WINPROP_MS_POLARIZATION_MINUS45 : An enum constant representing the minus 45 degree polarization
- WINPROP_MS_POLARIZATION_PLUS45 : An enum constant representing the plus 45 degree polarization
- WINPROP_MS_POLARIZATION_RHCP : An enum constant representing the RHCP polarization
- WINPROP_MS_POLARIZATION_LHCP : An enum constant representing the LHCP polarization
- WINPROP_MS_POLARIZATION_X_POL : An enum constant representing the X polarization

WINPROP_MS_ARRAY_TYPE

- WINPROP_MS_ARRAY_SINGLE_ANTENNA : An enum constant representing an array with a single antenna
- WINPROP_MS_ARRAY_INDIVIDUAL : An enum constant representing an array with individually placed elements
- WINPROP_MS_ARRAY_CIRCULAR : An enum constant representing an array with circular placed elements
- WINPROP_MS_ARRAY_LINEAR : An enum constant representing an array with elements placed separated within one dimension
- WINPROP_MS_ARRAY_RECTANGULAR : An enum constant representing an array with elements placed separated within two dimensions

WINPROP_MS_CHANNEL_TYPE

- WINPROP_MS_CHANNEL_PROPAGATION : An enum constant representing channel data amplitudes and phases determined from ray data
- WINPROP_MS_CHANNEL_RANDOM : An enum constant representing channel data amplitudes from ray data, gaussian distributed phases

WINPROP_MS_SNIR_TYPE

- WINPROP_MS_SNIR_MEAN : An enum constant representing mean SNIR
- WINPROP_MS_SNIR_CALC : An enum constant representing SNIR to be computed

WINPROP_MS_NORMALIZE_TYPE

- WINPROP_MS_NORMALIZE_NONE : An enum constant representing no normalization
- WINPROP_MS_NORMALIZE_TIME_FROBENIUS : An enum constant representing Froebenius normalization in time domain
- WINPROP_MS_NORMALIZE_TIME_ENERGY_CHANNEL : An enum constant representing normalization of channel energy in time domain
- WINPROP_MS_NORMALIZE_TIME_ENERGY_SUBCHANNEL : An enum constant representing normalization of sub-channel energy in time domain
- WINPROP_MS_NORMALIZE_TIME_STRONGEST_RAY : An enum constant representing normalization to strongest ray in time domain
- WINPROP_MS_NORMALIZE_FREQ_FROBENIUS : An enum constant representing Froebenius normalization in frequency domain
- WINPROP_MS_NORMALIZE_FREQ_ENERGY_CHANNEL : An enum constant representing normalization of channel energy in frequency domain
- WINPROP_MS_NORMALIZE_FREQ_PATHLOSS : An enum constant representing normalization of path loss in frequency domain
- WINPROP_MS_NORMALIZE_FREQ_SNIR : An enum constant representing normalization of SNIR in frequency domain

WINPROP_MS_CHANNEL_MATRIX_MODE

- WINPROP_MS_CHANNEL_MATRIX_REAL_IMAG : An enum constant representing the channel matrix elements to be in format real/imag
- WINPROP_MS_CHANNEL_MATRIX_MAG_PHASE : An enum constant representing the channel matrix elements to be in format real/imag

WINPROP_MS_CHANNEL_CAPACITY_MODE

- WINPROP_MS_CHANNEL_CAPACITY_EQUAL : An enum constant representing the power allocation (equal power here) scheme for channel capacity computation.
- WINPROP_MS_CHANNEL_CAPACITY_WATER_FILLING : An enum constant representing the power allocation (water filling here) scheme for channel capacity computation.

The documentation was generated from the following file:

- `source/SuperposeMS/winprop_superposems_types.h`

2.4.3 Initialisation Functions

Function List

WinProp_Structure_Init_MS_AdditionalResults
: This function initialises WinProp_MS_AdditionalResults struct with reasonable values

WinProp_Structure_Init_MS_Para
: This function initialises WinProp_MS_AdditionalResults struct with reasonable values

Function Details

int ***WinProp_Structure_Init_MS_AdditionalResults****(*WinProp_MS_AdditionalResults * *toInit)*

Description

This function initialises WinProp_MS_AdditionalResults struct with reasonable values

Parameters

WinProp_MS_AdditionalResults * *toInit*
: struct to initialise.

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_Structure_Init_MS_Para****(*WinProp_MS_Para * *toInit)*

Description

This function initialises WinProp_MS_AdditionalResults struct with reasonable values

Parameters

WinProp_MS_Para * *toInit*
: struct to initialise.

Returns An integer: 0 = success, failure otherwise.

The documentation was generated from the following file:

- `source/SuperposeMS/winprop_superposems_engine.h`

2.5 WinProp Network Planning Interface

Network planning interface

This chapter contains the following:

1. Network planning functions
2. Defines and constants

2.5.1 Network planning functions

Function List

WinProp_Net_Project_Open
: This function creates a new network planning project

WinProp_Net_Project_Open_WST
: This function creates a new network planning project

WinProp_Net_Project_Close
: This function closes an existing project.

WinProp_Net_Project_Para_Get
: This function gets parameters of the current air interface

WinProp_Net_Project_Para_Set
: Set parameters of the current air interface

WinProp_Net_Project_Compute
: This function starts computation with a network module.

WinProp_Net_Project_ComputePoints
: Start computation with a network module on points.

WinProp_Net_Project_ComputeTrajectories
: Start computation with a network module on trajectories.

WinProp_Net_PropagationMap_Add
: Add wave propagation result.

WinProp_Net_LosMap_Add
: Add Line-of-sight result

WinProp_Net_RayMatrix_Add
: Add ray matrix result

WinProp_Net_RayMatrixPlane_Add
: Add ray matrix results for planes

WinProp_Net_PropagationPlane_Add
: Add wave propagation result on planes.

WinProp_Net_PropagationPoint_Add
: Add wave propagation result of a list of points

WinProp_Net_PropagationTrajectory_Add
: Add wave propagation result of a list of trajectories

WinProp_Net_PropagationMap_Delete
: Delete wave propagation results.

WinProp_Net_PropagationPlane_Delete
: Delete wave propagation results in a plane.

WinProp_Net_PropagationPoints_Delete
: Delete wave propagation point results.

WinProp_Net_PropagationTrajectory_Delete
: Delete wave propagation trajectory results.

WinProp_Net_PropagationMap_Clean
: Remove all wave propagation results.

WinProp_Net_NetworkMap_Global_Get
: Retrieve computed network results regardless of transmission mode or carrier.

WinProp_Net_NetworkMap_Get
: Obtain computed network results.

WinProp_Net_NetworkMap_Carrier_Get
: Obtain computed network results individually for a carrier.

WinProp_Net_NetworkMonteCarloStats_Get
: Gets the Monte-Carlo result statistics of a network simulation.

WinProp_Net_NetworkPlane_Global_Get
: Retrieve computed network results on planes regardless of transmission mode or carrier.

WinProp_Net_NetworkPlane_Get
: Obtain computed network result on planes.

WinProp_Net_NetworkPlane_Carrier_Get
: Obtain a individually for a carrier computed network result.

WinProp_Net_NetworkTrajectoryList_Get
: Obtain computed network trajectories result.

WinProp_Net_NetworkTrajectoryList_Carrier_Get
: Obtain a individually for a carrier computed network trajectories result.

WinProp_Net_PropagationMap_GetTrxInfo
: Get information about transmitter.

WinProp_Net_Carrier_Add
: Add carrier to air interface.

WinProp_Net_Carrier_Check
: Get information about the carrier.

WinProp_Net_Carrier_Delete
: Delete a carrier from interface.

WinProp_Net_Carrier_Para_Set
: Sets a parameter of a carrier

WinProp_Net_Carrier_Para_Get
: Gets a parameter of a carrier.

WinProp_Net_TransmissionMode_Add
: Add new MCS/transmission mode/service.

WinProp_Net_TransmissionMode_Delete
: Delete existing MCS/transmission mode.

WinProp_Net_TransmissionMode_Para_Set
: Modify parameter of MCS.

WinProp_Net_TransmissionMode_Para_Get
: Retrieve information about a network planning parameter.

WinProp_Net_Channel_Add
: Add new channel to network project.

WinProp_Net_Channel_Para_Set
: Set parameter of a channel.

WinProp_Net_Channel_Para_Get
: Get parameter of a channel.

WinProp_Net_Channel_Delete
: Delete channel from network project

WinProp_Net_PrioMap_Set
: Set priority map for optimizer.

WinProp_Net_PrioMap_Get
: Get priority map for optimizer.

WinProp_Net_Application_Add
: Adds an application to the network planning project.

WinProp_Net_Application_Delete
: Deletes an application from the network planning project.

WinProp_Net_Application_TransmissionMode_Link
: Links an application to a transmission mode and sets the preference of that link.

WinProp_Net_Application_TransmissionMode_Unlink
: Unlinks a transmission mode from an application.

WinProp_Net_Application_ClutterClass_Link
: Links an application to a clutter class and sets various parameters of this link.

WinProp_Net_Application_ClutterClass_Unlink
: Unlinks a clutter class from an application.

WinProp_Net_Application_Para_Set
: Sets a parameter of an application.

WinProp_Net_Application_Para_Get
: Gets a parameter of an application.

WinProp_Net_Clutter_Set
: Sets the clutter data needed for network planning projects with Monte-Carlo simulation.

WinProp_Net_Project_Result_Switch
: Sets the computation of the specified result type to the specified state.

WinProp_Net_Project_Result_Enable
: Enables the computation of the specified result type.

WinProp_Net_Project_Result_Disable
: Disables the computation of the specified result type.

Function Details

int ***WinProp_Net_Project_Open****(int * OUT_Handle, unsigned int IN_AirInterface,* WinProp_Callback ** IN_Callback)*

Description

This function creates a new network planning project

Parameters

*int * OUT_Handle*
: Handle of this project.

unsigned int IN_AirInterface
: Type of air interface. see defines in "AI_Define.h".

WinProp_Callback * *IN_Callback*
: Pointer to callback functions for simulation progress.

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_Net_Project_Open_WST****(int * OUT_Handle, const char * IN_AirInterface,* WinProp_Callback ** IN_Callback)*

Description

This function creates a new network planning project

Parameters

*int * OUT_Handle*
: Handle of this project.

*const char * IN_AirInterface*
: Air interface description file.

WinProp_Callback * *IN_Callback*
: Pointer to callback functions for simulation progress.

Returns An integer : 0 = success, failure otherwise.

int ***WinProp_Net_Project_Close****(int IN_Handle)*

Description

This function closes an existing project.

Parameters

int IN_Handle
Handle of project to be closed.

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_Net_Project_Para_Get****(int IN_Handle, int IN_Parameter, double * OUT_ReturnDouble, int * OUT_ReturnInt, char * OUT_ReturnChar)*

Description

This function gets parameters of the current air interface

Parameters

int IN_Handle
Handle of the project.

int IN_Parameter
Parameter to be retreived.

*double * OUT_ReturnDouble*
Return value (if double required)

*int * OUT_ReturnInt*
Return value (if int required)

*char * OUT_ReturnChar*
Return value (if string required)

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_Net_Project_Para_Set****(int IN_Handle, int IN_Parameter, const double * IN_Double, const int * IN_Int, const char * IN_Char)*

Description

Set parameters of the current air interface

Parameters

int IN_Handle
Handle of the project.

int IN_Parameter
Parameter to be set.

*const double * IN_Double*
Return value (if double required)

*const int * IN_Int*
Return value (if int required)

*const char * IN_Char*
Return value (if string required)

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_Net_Project_Compute****(int IN_Handle,* WinProp_Callback ** IN_Callback)*

Description

This function starts computation with a network module.

Parameters

int IN_Handle
Handle of the project to be computed.

WinProp_Callback * *IN_Callback*
Pointer to the callback functions for simulation progress.

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_Net_Project_ComputePoints****(int IN_Handle,* WinProp_Callback ** IN_Callback)*

Description

Start computation with a network module on points.

Parameters

int IN_Handle
Handle of the project to be computed.

WinProp_Callback * *IN_Callback*
Pointer to the callback functions for simulation progress.

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_Net_Project_ComputeTrajectories****(int IN_Handle,* WinProp_Callback ** IN_Callback)*

Description

Start computation with a network module on trajectories.

Parameters

int IN_Handle
Handle of the project to be computed.

WinProp_Callback * *IN_Callback*
Pointer to the callback functions for simulation progress.

Returns An integer : 0 = success, failure otherwise.

int ***WinProp_Net_PropagationMap_Add****(int IN_Handle, int * OUT_MapIndex,* WinProp_Antenna ** IN_Transmitter,* WinProp_Result ** IN_Map)*

Description

Add wave propagation result.

Parameters

int IN_Handle
Handle of the project.

*int * OUT_MapIndex*
Assigned index to this map.

WinProp_Antenna * *IN_Transmitter*
Information about transmitter.

WinProp_Result * *IN_Map*
Wave propagation map.

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_Net_LosMap_Add****(int IN_Handle, int * OUT_MapIndex,* WinProp_Antenna * *IN_Transmitter,* WinProp_Result * *IN_LosMap)*

Description

Add Line-of-sight result

Parameters

int IN_Handle
Handle of the project.

*int * OUT_MapIndex*
Assigned index to this map.

WinProp_Antenna * *IN_Transmitter*
Information about transmitter.

WinProp_Result * *IN_LosMap*
LOS result map.

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_Net_RayMatrix_Add****(int IN_Handle, int * OUT_MapIndex,* WinProp_Antenna * *IN_Transmitter,* WinProp_RayMatrix * *IN_RayMatrix)*

Description

Add ray matrix result

Parameters

int IN_Handle
Handle of the project.

*int * OUT_MapIndex*
Assigned index to this map.

WinProp_Antenna * *IN_Transmitter*
Information about transmitter.

WinProp_RayMatrix * *IN_RayMatrix*
ray matrix result.

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_Net_RayMatrixPlane_Add****(int IN_Handle, int * OUT_MapIndex,* WinProp_Antenna * *IN_Transmitter,* WinProp_RayMatrixList * *IN_RayMatriList)*

Description

Add ray matrix results for planes

Parameters

int IN_Handle
Handle of the project.

*int * OUT_MapIndex*
Assigned index to this map.

WinProp_Antenna * *IN_Transmitter*
Information about transmitter.

WinProp_RayMatrixList * *IN_RayMatriList*
ray matrix results.

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_Net_PropagationPlane_Add****(int IN_Handle, int * OUT_MapIndex,* WinProp_Antenna * *IN_Transmitter,* WinProp_ResultPlaneList * *IN_MapPlanes)*

Description

Add wave propagation result on planes.

Parameters

int IN_Handle
Handle of the project.

*int * OUT_MapIndex*
Assigned index to this map

WinProp_Antenna * *IN_Transmitter*
Information about transmitter.

WinProp_ResultPlaneList * *IN_MapPlanes*
Wave propagation map (planes).

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_Net_PropagationPoint_Add****(int IN_Handle, int * OUT_MapIndex,* WinProp_Antenna * *IN_Transmitter,* WinProp_ResultPointsList * *IN_MapPoints)*

Description

Add wave propagation result of a list of points

Parameters

int IN_Handle
Handle of the project.

*int * OUT_MapIndex*
Assigned index to this map.

WinProp_Antenna * *IN_Transmitter*
Information about transmitter.

WinProp_ResultPointsList * *IN_MapPoints*
Wave propagation points list.

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_Net_PropagationTrajectory_Add****(int IN_Handle, int * OUT_MapIndex, const* WinProp_Antenna * *IN_Transmitter, const* WinProp_Trajectory * *IN_Trajectories, int IN_NrTrajectories, const* WinProp_ResultTrajectoryList * *IN_MapTrajs)*

Description

Add wave propagation result of a list of trajectories

Parameters

int IN_Handle
Handle of the project.

*int * OUT_MapIndex*
Assigned index to this map.

const WinProp_Antenna * *IN_Transmitter*
Information about transmitter.

const WinProp_Trajectory * *IN_Trajectories*
Information about receiver configuration in trajectory mode.

int IN_NrTrajectories
Number of trajectories.

const WinProp_ResultTrajectoryList * *IN_MapTrajs*
Wave propagation result list of trajectories.

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_Net_PropagationMap_Delete****(int IN_Handle, int IN_MapIndex)*

Description

Delete wave propagation results.

Parameters

int IN_Handle
Handle of the project.

int IN_MapIndex
Index of map to be removed.

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_Net_PropagationPlane_Delete****(int IN_Handle, int IN_MapIndex)*

Description

Delete wave propagation results in a plane.

Parameters

int IN_Handle
Handle of the project.

int IN_MapIndex
Index of map to be removed.

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_Net_PropagationPoints_Delete****(int IN_Handle, int IN_MapIndex)*

Description

Delete wave propagation point results.

Parameters

int IN_Handle
Handle of the project.

int IN_MapIndex
Index of map to be removed.

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_Net_PropagationTrajectory_Delete****(int IN_Handle, int IN_MapIndex)*

Description

Delete wave propagation trajectory results.

Parameters

int IN_Handle
Handle of the project.

int IN_MapIndex
Index of map to be removed.

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_Net_PropagationMap_Clean****(int IN_Handle)*

Description

Remove all wave propagation results.

Parameters

int IN_Handle
Handle of the project.

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_Net_NetworkMap_Global_Get****(int IN_Handle, int IN_ResultType,* WinProp_Result ** *OUT_NetworkResult)*

Description

Retrieve computed network results regardless of transmission mode or carrier.

Parameters

int IN_Handle
Handle of project.

int IN_ResultType
Type of result to be retrieved.

WinProp_Result ** *OUT_NetworkResult*
Pointer to network result map.

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_Net_NetworkMap_Get****(int IN_Handle, int IN_Service, int IN_ResultType,* WinProp_Result ** *OUT_NetworkResult)*

Description

Obtain computed network results.

Parameters

int IN_Handle
Handle of project.

int IN_Service
Index of service or application in case of a Monte-Carlo area result.

int IN_ResultType
Type of result to be retrieved.

WinProp_Result ** *OUT_NetworkResult*
Pointer to network result map.

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_Net_NetworkMap_Carrier_Get****(int IN_Handle, int IN_Service, int IN_Carrier, int IN_ResultType,* WinProp_Result ** *OUT_NetworkResult)*

Description

Obtain computed network results individually for a carrier.

Parameters

int IN_Handle
Handle of project.

int IN_Service
Index of service or application in case of a Monte-Carlo area result.

int IN_Carrier
Index of carrier.

int IN_ResultType
Type of result to be retrieved.

WinProp_Result ** *OUT_NetworkResult*
Pointer to network result map.

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_Net_NetworkMonteCarloStats_Get****(int IN_Handle,* WinProp_ResultMonteCarlo ** *OUT_NetworkResult)*

Description

Gets the Monte-Carlo result statistics of a network simulation.

Parameters

int IN_Handle
Handle of the project.

WinProp_ResultMonteCarlo ** *OUT_NetworkResult*
If non-null, the out network result.

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_Net_NetworkPlane_Global_Get****(int IN_Handle, int IN_ResultType,* WinProp_ResultPlaneList ** *OUT_NetworkPlanes)*

Description

Retrieve computed network results on planes regardless of transmission mode or carrier.

Parameters

int IN_Handle
Handle of project.

int IN_ResultType
Type of result to be retrieved.

WinProp_ResultPlaneList ** *OUT_NetworkPlanes*
Pointer to network result map.

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_Net_NetworkPlane_Get****(int IN_Handle, int IN_Service, int IN_ResultType,* WinProp_ResultPlaneList ** *OUT_NetworkPlanes)*

Description

Obtain computed network result on planes.

Parameters

int IN_Handle
Handle of project.

int IN_Service
Index of service or application in case of a Monte-Carlo area result.

int IN_ResultType
Type of result to be retrieved.

WinProp_ResultPlaneList ** *OUT_NetworkPlanes*
Pointer to network result map.

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_Net_NetworkPlane_Carrier_Get****(int IN_Handle, int IN_Service, int IN_Carrier, int IN_ResultType,* WinProp_ResultPlaneList *** OUT_NetworkPlanes)*

Description

Obtain a individually for a carrier computed network result.

Parameters

int IN_Handle
Handle of the project.

int IN_Service
Index of service or application in case of a Monte-Carlo area result.

int IN_Carrier
Carrier.

int IN_ResultType
Type of result to be retrieved.

WinProp_ResultPlaneList *** OUT_NetworkPlanes*
Pointer to network result map.

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_Net_NetworkTrajectoryList_Get****(int IN_Handle, int IN_Service, int IN_ResultType,* WinProp_ResultTrajectoryList *** OUT_NetworkTrajectoiries)*

Description

Obtain computed network trajectories result.

Parameters

int IN_Handle
Handle of project.

int IN_Service
Index of service or application in case of a Monte-Carlo area result.

int IN_ResultType
Type of result to be retrieved.

WinProp_ResultTrajectoryList *** OUT_NetworkTrajectoiries*
Pointer to network trajectory list result map.

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_Net_NetworkTrajectoryList_Carrier_Get****(int IN_Handle, int IN_Service, int IN_Carrier, int IN_ResultType,* WinProp_ResultTrajectoryList *** OUT_NetworkTrajectoiries)*

Description

Obtain a individually for a carrier computed network trajectories result.

Parameters

int IN_Handle
Handle of the project.

int IN_Service
Index of service or application in case of a Monte-Carlo area result.

int IN_Carrier
Carrier.

int IN_ResultType
Type of result to be retrieved.

WinProp_ResultTrajectoryList ** *OUT_NetworkTrajectoiries*
Pointer to network trajectory list result map.

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_Net_PropagationMap_GetTrxInfo****(int IN_Handle, int IN_MapIndex,* WinProp_Antenna ** *OUT_Transmitter)*

Description

Get information about transmitter.

Parameters

int IN_Handle
Handle of the project.

int IN_MapIndex
Assigned index to this map.

WinProp_Antenna ** *OUT_Transmitter*
Information about transmitter .

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_Net_Carrier_Add****(int IN_Handle, unsigned int IN_CarrierIndex)*

Description

Add carrier to air interface.

Parameters

int IN_Handle
Handle of the project.

unsigned int IN_CarrierIndex
Index of new carrier.

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_Net_Carrier_Check****(int IN_Handle, unsigned int IN_CarrierIndex, double * OUT_FrequencyUplink, double * OUT_FrequencyDownlink)*

Description

Get information about the carrier.

Parameters

int IN_Handle
Handle of the project.

unsigned int IN_CarrierIndex
Index of the carrier.

*double * OUT_FrequencyUplink*
Uplink (UL) frequency of the carrier in MHz.

*double * OUT_FrequencyDownlink*
Downlink (DL) frequency of the carrier in MHz.

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_Net_Carrier_Delete****(int IN_Handle, unsigned int IN_CarrierIndex)*

Description

Delete a carrier from interface.

Parameters

int IN_Handle
Handle of the project.

unsigned int IN_CarrierIndex
Index of carrier to be removed.

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_Net_Carrier_Para_Set****(int IN_Handle, int IN_Index, int IN_Parameter, double * IN_DoubleValue, int * IN_IntValue, char * IN_Char)*

Description

Sets a parameter of a carrier

Parameters

int IN_Handle
Handle of project.

int IN_Index
ID of carrier.

int IN_Parameter
The parameter to be set.

*double * IN_DoubleValue*
Double (if required)

*int * IN_IntValue*
Integer (if required)

*char * IN_Char*
String (if required)

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_Net_Carrier_Para_Get****(int IN_Handle, int IN_Index, int IN_Parameter, double * OUT_RetuernDoubleValue, int * OUT_RetuernIntValue, char * OUT_RetuernCharValue)*

Description

Gets a parameter of a carrier.

Parameters

int IN_Handle
Handle of project.

int IN_Index
ID of carrier.

int IN_Parameter
The parameter to get.

*double * OUT_RetuernDoubleValue*
If non-null, the return double value.

*int * OUT_RetuernIntValue*
If non-null, the return int value.

*char * OUT_RetuernCharValue*
If non-null, the return character.

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_Net_TransmissionMode_Add****(int IN_Handle, char * IN_Name, unsigned int IN_Index)*

Description

Add new MCS/transmission mode/service.

Parameters

int IN_Handle
Handle of the project.

*char * IN_Name*
Name of MCS.

unsigned int IN_Index
Index of new MCS.

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_Net_TransmissionMode_Delete****(int IN_Handle, unsigned int IN_Index)*

Description

Delete existing MCS/transmission mode.

Parameters

int IN_Handle
Handle of the project.

unsigned int IN_Index
Index of MCS to be deleted.

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_Net_TransmissionMode_Para_Set****(int IN_Handle, int IN_Index, int IN_Parameter, double * IN_DoubleValue, int * IN_IntValue, char * IN_Char)*

Description

Modify parameter of MCS.

Parameters

int IN_Handle
Handle of the project.

int IN_Index
Index of the MCS.

int IN_Parameter
Parameter to modify.

*double * IN_DoubleValue*
Double (if required)

*int * IN_IntValue*
Integer (if required)

*char * IN_Char*
String (if required)

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_Net_TransmissionMode_Para_Get****(int IN_Handle, int IN_Index, int IN_Parameter, double * OUT_ReturnDoubleValue, int * OUT_ReturnIntValue, char * OUT_ReturnChar)*

Description

Retrieve information about a network planning parameter.

Parameters

int IN_Handle
Handle of the project.

int IN_Index
Index of MCS.

int IN_Parameter
Parameter to retrieve.

*double * OUT_ReturnDoubleValue*
Double (if required)

*int * OUT_ReturnIntValue*
Integer (if required)

*char * OUT_ReturnChar*
String (if required)

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_Net_Channel_Add****(int IN_Handle, char * IN_Name, unsigned int * OUT_Index)*

Description

Add new channel to network project.

Parameters

int IN_Handle
Handle of the project.

*char * IN_Name*
Name of new channel.

*unsigned int * OUT_Index*
Index of new channel.

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_Net_Channel_Para_Set****(int IN_Handle, int IN_Index, int IN_Parameter, double * IN_DoubleValue, int * IN_IntValue, char * IN_Char)*

Description

Set parameter of a channel.

Parameters

int IN_Handle
Handle of the project.

int IN_Index
Index of channel.

int IN_Parameter
Parameter to modify.

*double * IN_DoubleValue*
Double (if required)

*int * IN_IntValue*
Integer (if required)

*char * IN_Char*
String (if required)

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_Net_Channel_Para_Get****(int IN_Handle, int IN_Index, int IN_Parameter, double * OUT_DoubleValue, int * OUT_IntValue, char * OUT_Char)*

Description

Get parameter of a channel.

Parameters

int IN_Handle
Handle of the project.

int IN_Index
Index of channel.

int IN_Parameter
Parameter to retrieve.

*double * OUT_DoubleValue*
Double (if required)

*int * OUT_IntValue*
Integer (if required)

*char * OUT_Char*
String (if required)

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_Net_Channel_Delete****(int IN_Handle, unsigned int IN_Index)*

Description

Delete channel from network project

Parameters

int IN_Handle
Handle of the project.

unsigned int IN_Index
Index of channel to delete.

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_Net_PrioMap_Set****(int IN_Handle,* WinProp_Result ** IN_PrioMap)*

Description

Set priority map for optimizer.

Parameters

int IN_Handle
Handle of the project.

WinProp_Result ** IN_PrioMap*
Priority map.

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_Net_PrioMap_Get****(int IN_Handle,* WinProp_Result *** OUT_PrioMap)*

Description

Get priority map for optimizer.

Parameters

int IN_Handle
Handle of the project.

WinProp_Result ** *OUT_PrioMap*
Priority map.

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_Net_Application_Add****(int IN_Handle, int IN_ID, int IN_Priority, float IN_Activity, int IN_TrafficMode, const char * IN_Name)*

Description

Adds an application to the network planning project.

Parameters

int IN_Handle
Handle of the project.

int IN_ID
ID of the application.

int IN_Priority
Application priority.

float IN_Activity
Application activity.

int IN_TrafficMode
The traffic mode of the application.

*const char * IN_Name*
Name of the application.

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_Net_Application_Delete****(int IN_Handle, int IN_ID)*

Description

Deletes an application from the network planning project.

Parameters

int IN_Handle
Handle of project.

int IN_ID
ID of application.

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_Net_Application_TransmissionMode_Link****(int IN_Handle, int IN_ID, int IN_TransmissionModeID, int IN_Preference)*

Description

Links an application to a transmission mode and sets the preference of that link.

Parameters

int IN_Handle
Handle of the project.

int IN_ID
ID of application.

int IN_TransmissionModeID
Identifier for the transmission mode.

int IN_Preference
The preference.

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_Net_Application_TransmissionMode_Unlink****(int IN_Handle, int IN_ID, int IN_TransmissionModeID)*

Description

Unlinks a transmission mode from an application.

Parameters

int IN_Handle
Handle of project.

int IN_ID
ID of application.

int IN_TransmissionModeID
Identifier for the transmission mode.

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_Net_Application_ClutterClass_Link****(int IN_Handle, int IN_ID, int IN_ClutterID, float IN_ArrivalRate, float IN_HoldTime, float IN_OfferedTraffic)*

Description

Links an application to a clutter class and sets various parameters of this link.

Parameters

int IN_Handle
Handle of project.

int IN_ID
ID of application.

int IN_ClutterID
Identifier for the clutter class.

float IN_ArrivalRate
The arrival rate.

float IN_HoldTime
The hold time.

float IN_OfferedTraffic
The offered traffic.

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_Net_Application_ClutterClass_Unlink****(int IN_Handle, int IN_ID, int IN_ClutterID)*

Description

Unlinks a clutter class from an application.

Parameters

int IN_Handle
Handle of project.

int IN_ID
ID of application.

int IN_ClutterID
Identifier for the clutter class.

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_Net_Application_Para_Set****(int IN_Handle, int IN_Index, int IN_Parameter, const double * IN_DoubleValue, const int * IN_IntValue, const char * IN_CharValue)*

Description

Sets a parameter of an application.

Parameters

int IN_Handle
Handle of project.

int IN_Index
ID of application.

int IN_Parameter
The parameter to be set.

*const double * IN_DoubleValue*
Double (if required)

*const int * IN_IntValue*
Integer (if required)

*const char * IN_CharValue*
String (if required)

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_Net_Application_Para_Get****(int IN_Handle, int IN_Index, int IN_Parameter, double * OUT_ReturnDoubleValue, int * OUT_ReturnIntValue, char * OUT_ReturnChar)*

Description

Gets a parameter of an application.

Parameters

int IN_Handle
Handle of project.

int IN_Index
ID of application.

int IN_Parameter
The parameter to get.

*double * OUT_ReturnDoubleValue*
If non-null, the return double value.

*int * OUT_ReturnIntValue*
If non-null, the return int value.

*char * OUT_ReturnChar*
If non-null, the return character.

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_Net_Clutter_Set****(int IN_Handle, int IN_ClutterMode, const char * IN_ClutterTraffic, bool IN_ClutterIsIndex, const char * IN_ClutterTable,* CLUTTER ** IN_ClutterData)*

Description

Sets the clutter data needed for network planning projects with Monte-Carlo simulation.

Parameters

int IN_Handle
Handle of the project.

int IN_ClutterMode
The clutter mode.

- CLUTTERMODE_NONE = don't use clutter data
- CLUTTERMODE_FILE = load clutter data from specified files
- CLUTTERMODE_RAM = use clutter data from memory

*const char * IN_ClutterTraffic*
Filename to the clutter traffic database.

bool IN_ClutterIsIndex
True if clutter traffic is an index database.

*const char * IN_ClutterTable*
Filename to the clutter table.

CLUTTER ** IN_ClutterData*
If non-null, clutter data stored in memory, IN_ClutterMode needs to be set to CLUTTERMODE_RAM.

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_Net_Project_Result_Switch****(int IN_Handle, int IN_Parameter, bool IN_State)*

Description

Sets the computation of the specified result type to the specified state.

Parameters

int IN_Handle
Handle of project.

int IN_Parameter
The result type.

bool IN_State
Enable / disable the computation.

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_Net_Project_Result_Enable****(int IN_Handle, int IN_Parameter)*

Description

Enables the computation of the specified result type.

Parameters

int IN_Handle
Handle of project.

int IN_Parameter
The result type.

Returns An int.

int ***WinProp_Net_Project_Result_Disable****(int IN_Handle, int IN_Parameter)*

Description

Disables the computation of the specified result type.

Parameters

int IN_Handle
Handle of project.

int IN_Parameter
The result type.

Returns An int.

The documentation was generated from the following file:

- `source/Net/Net.h`

2.5.2 Defines and constants

Detailed Description

Network planning interface defines. These are used to set/retrieve various parameters of network planning through the functions WinProp_Net_Project_Para_Set and WinProp_Net_Project_Para_Get.

Generic air interfaces

NET_AIRINTERFACE_GSM_GENERIC *0x0000*
Generic GSM standard air interface.

NET_AIRINTERFACE_TETRA_GENERIC *0x0020*
Generic TETRA standard air interface.

NET_AIRINTERFACE_802_11_A_GENERIC *0x0040*
Generic IEEE 802.11a standard air interface.

NET_AIRINTERFACE_802_11_G_GENERIC *0x0080*
Generic IEEE 802.11g standard air interface.

NET_AIRINTERFACE_802_11_N_GENERIC *0x00A0*
Generic IEEE 802.11n standard air interface.

NET_AIRINTERFACE_802_11_P_GENERIC *0x00C0*
Generic IEEE 802.11p standard air interface.

NET_AIRINTERFACE_802_16_2004_GENERIC *0x00E0*
Generic IEEE 802.16-2004 standard air interface.

NET_AIRINTERFACE_802_16_E_GENERIC *0x0100*
Generic IEEE 802.16e standard air interface.

NET_AIRINTERFACE_LTE_GENERIC *0x0140*
Generic LTE standard air interface.

NET_AIRINTERFACE_CDMA_GENERIC *0x0170*
Generic CDMA standard air interface.

NET_AIRINTERFACE_5G_FDD_GENERIC *0x0180*
Generic 5G-FDD standard air interface.

NET_AIRINTERFACE_5G_TDD_GENERIC *0x0190*
Generic 5G-TDD standard air interface.

Common parameters for different air interfaces

NET_PARA_PREDEFINED_AIRINTERFACE *-4*
Set predefined Air Interface.

NET_PARA_MULTIPLE_ACCESS *-1*
Multiple Access Mode.

NET_PARA_DUPLEX_MODE *-2*
Duplex Mode.

NET_PARA_DUPLEX_SUBMODE *-3*
Duplex Sub Mode.

NET_PARA_BANDWIDTH *0*
Bandwidth for LTE.

NET_PARA_CARRIER_SEPARATION *1*
Separation between carriers.

NET_PARA_DL_UL_SEPARATION *4*
Separation between UL and DL.

NET_PARA_ADJCPR *5*

NET_PARA_ALTCPR *6*

NET_PARA_ALLCPR *7*

NET_PARA_TIMESLOTS *8*
Number of time slots.

NET_PARA_TIMESLOTS_PER_USER *9*
Number of time slots per user.

NET_PARA_CHIPRATE *10*
Chip rate.

NET_PARA_ORTHOGONALITY_DL *11*
Orthogonality factor (Downlink).

NET_PARA_COMMON_CHANNEL_POWER *12*

NET_PARA_CARRIERS *13*
Number of carriers.

NET_PARA_MS_BODYLOSS *14*
Damping losses by the body of the user.

NET_PARA_MS_GAIN *15*
Mobile station antenna gain.

NET_PARA_TIMESLOTS_FURTHER *16*
Number of further time slots.

NET_PARA_MS_NOISEFIGURE *17*
Noise Figure Mobile Station.

NET_PARA_MS_FAST_FADING_MARGIN *18*
Fast Fading Margin.

NET_PARA_MS_MAX_TX_POWER *19*
Power Mobile Station (dBm).

NET_PARA_MS_POWER_CONTROL_RANGE *120*
Power control range from the maximum mobile station power (dB).

NET_PARA_SERVICES *20*
Number of services.

NET_PARA_RESOLUTION *21*
Resolution of result matrix.

NET_PARA_AREA_MODE *22*
Size of planning area: Automatic mode. Used in the WinProp_Net_Project_Para_Set function.

NET_PARA_AREA_LL_X *23*
X-coordinate of lower left corner of predicton area. Used in the WinProp_Net_Project_Para_Set function.

NET_PARA_AREA_LL_Y *24*
Y-coordinate of lower left corner of predicton area. Used in the WinProp_Net_Project_Para_Set function.

NET_PARA_AREA_UR_X *25*
X-coordinate of upper right corner of predicton area. Used in the WinProp_Net_Project_Para_Set function.

NET_PARA_AREA_UR_Y *26*
Y-coordinate of upper right corner of predicton area. Used in the WinProp_Net_Project_Para_Set function.

NET_PARA_CHANNELS *29*
Number of channels.

NET_PARA_FFT_SIZE *30*
FTT Size in LTE air interface. Used in the WinProp_Net_Project_Para_Set function.

NET_PARA_SUBCARRIERS_RESOURCE_BLOCK *32*
Number of subcarriers for one resource block in LTE air interface. Used in the WinProp_Net_Project_Para_Set function.

NET_PARA_CELL_ASSIGNMENT_MIN_REQ_SNIR *36*
Minimum required SNIR for cell assignment. Used in the WinProp_Net_Project_Para_Set function.

NET_PARA_CELL_ASSIGNMENT_MIN_REQ_POWER *37*
Minimum required power for cell assignment. Used in the WinProp_Net_Project_Para_Set function.

NET_PARA_CELL_ASSIGNMENT_SPREADING_FACTOR *38*
Spreading factor for cell assignment. Used in the WinProp_Net_Project_Para_Set function.

NET_PARA_CELL_ASSIGNMENT_MIN_REQ_POWER_USED *39*
Minimum required power used for Cell Assignment. Used in the WinProp_Net_Project_Para_Set function.

NET_PARA_HEIGHT *40*
Prediction height.

NET_PARA_ORTHOGONALITY *41*
Orthogonality factor.

NET_PARA_GUARD_PERIOD *45*
Guard Period.

NET_PARA_PROJECTNAME *46*
Name of project. Can be optionally set in WinProp_Net_Project_Para_Set.

NET_PARA_TDD_VARIABLE_SWITCH *47*
TDD Variable Switch.

NET_PARA_CELL_LOAD *48*
Cell Load.

NET_PARA_CELL_ASSIGNMENT_BACKOFF *50*
Backoff value for cell assignment.

Parameters for OFDM

NET_PARA_SAMPLING_RATE_FACTOR *51*
Sampling rate. Used in the WinProp_Net_Project_Para_Set.

NET_PARA_SUBCARRIERS_GUARD *52*
Sub-carrier guards.

NET_PARA_SUBCARRIERS_DC *53*
DC sub-carrier.

NET_PARA_SYMBOLS_MODE *54*
Symbol definition.

NET_PARA_SUBCARRIERS_CONTROL *55*
Sub-carriers for control. Used in the WinProp_Net_Project_Para_Set to set the number of control sub-carriers.

NET_PARA_SUBCARRIERS_REFERENCE *56*
Reference sub-carriers. Used in the WinProp_Net_Project_Para_Set to set the number of reference sub-carriers.

NET_PARA_SYMBOLS_CONTROL *57*
Control symbols. Used in the WinProp_Net_Project_Para_Set to set control symbols.

NET_PARA_SYMBOLS_REFERENCE *58*
Reference symbols. Used in the WinProp_Net_Project_Para_Set to set reference symbols.

NET_PARA_CELL_LOAD_MODE *59*
Cell Load Mode.

NET_PARA_HEIGHT_MULTIPLE *60*
Set multiple prediction heights.

NET_PARA_ASCII_OUTPUT *61*
Set additional ASCII output.

NET_PARA_BANDWIDTH_MODE_5G *62*
Set Bandwidth mode. Only relevant for 5G projects.

NET_PARA_INTERFERANCE_OVERLAP_RATIO *63*
Set Interference overlap ratio.

NET_PARA_POWER_BACKOFF_CONTROL *64*
Set Tx power backoff of control subcarriers.

NET_PARA_SAME_PILOT_SUBCARRIERS *65*
Use in each cell same subcarriers for pilot signals. Relevant if symbol mode is set to identical subcarriers for each symbol.

NET_PARA_GUARD_SYNCHRONIZATION *144*
Guard interval synchronization (trigger on first or strongest contribution).

NET_OFDM_GUARD_SYNC_FIRST *0*
Guard interval is synchronized to trigger on first contribution.

NET_OFDM_GUARD_SYNC_STRONGEST *1*
Guard interval is synchronized to trigger on strongest contribution.

Parameters for MIMO systems (LTE, 802.11n)

NET_PARA_MIMO_NUMBER_STREAMS *70*
Number of MIMO streams.

NET_PARA_MIMO_SIGNALLING_OVERHEAD *71*
MIMO: Overhead ratio.

NET_PARA_MIMO_BEAMFORMING_GAIN_SERVING *72*
MIMO: Beamforming serving signal.

NET_PARA_MIMO_BEAMFORMING_GAIN_INTERFERENCE *73*
MIMO: Beamforming interfering signal.

NET_PARA_MIMO_INTERFERENCE_MODE *74*
MIMO: Interference mode.

General Parameters

NET_PARA_TDD_TIME_SWITCH *92*
TDD switch.

NET_PARA_TDD_NBR_DL *93*
TDD Number DL (downlink).

NET_PARA_TDD_NBR_UL *94*
TDD Number UL (uplink).

NET_PARA_TDD_NBR_SWITCH *95*
TDD Number Switch.

NET_PARA_TDD_SWITCH_DL *96*
TDD Number Switch DL (downlink).

NET_PARA_TDD_SWITCH_UL *97*
TDD Number Switch UL (downlink).

NET_PARA_TDD_SWITCH_GP *98*
TDD Switch Ratio GP.

NET_PARA_GUARD_TIME *99*
Guard time.

NET_MAX_USERS_PER_CARRIER *100*
Maximum number of users per carrier.

NET_WINDOW_SOFT_HANDOVER *101*
Soft Handover Window.

NET_WINDOW_SOFTER_HANDOVER *102*
Softer Handover Window.

NET_PARA_TRAFFIC_MODE *103*
Monte carlo traffic mode.

NET_PARA_APPLICATION_AREA_MODE *104*
Prediction area units (Kilometres = NET_APPLICATION_AREA_MODE_KILOMETER / metres = NET_APPLICATION_AREA_MODE_METER

NET_PARA_TRAFFIC_MONTE_CARLO_LOOPS *105*
Number of Monte Carlo runs.

NET_PARA_TRAFFIC_MONTE_CARLO_DRAW *106*
Monte carlo parameter. The fourth parameter of WinProp_Net_Project_Para_Set can be either 1 or 0.

NET_PARA_SIMULATE_EACH_CARRIER *107*
Decision variable to simulate all carriers. The fourth parameter of WinProp_Net_Project_Para_Set can be either 1 or 0.

NET_PARA_TRAFFIC_CELL_SELECT_CONSIDER_PRIO *108*
Decision variable to consider priority of an application in a cell. The fourth parameter of WinProp_Net_Project_Para_Set can be set to 1 or 0.

NET_PARA_TRAFFIC_CELL_SELECT_UPDATE_LIST *109*
Update cell list with new value. The fourth parameter of WinProp_Net_Project_Para_Set can be set to NET_CELL_SELECT_UPDATE_NEW_PATH_LOSSES or NET_CELL_SELECT_UPDATE_NEW_LOAD_VALUE.

NET_PARA_TRAFFIC_CELL_SELECT_ALGORITHM *110*
Select user algorithm.

NET_PARA_HEIGHT_ADD *111*
Add prediction height.

NET_PARA_TDMA_GUARD_INTERVAL *112*
TDMA guard interval between time slots.

NET_PARA_CELL_NOISE_RISE *113*
Set noise rise in uplink due to other interfering mobile stations.

NET_PARA_SYMBOLS_DATA *114*
Set number of data symbols (time).

NET_PARA_TRAFFIC_MAX_ITERATIONS *115*
Set Max number of iterations for traffic simulations.

NET_PARA_TRAFFIC_CELL_LOAD_THRESHOLD *116*
Set threshold for changes of cell.

NET_PARA_REPRO_NUMBER *117*
Initialisation of random processes.

NET_SETTINGS_NUMBER_THREADS *118*

NET_SETTINGS_SAVE_FILTER *119*

NET_PARA_REPEATERS_MULTIHOPS *145*
Set number of repeaters multihops.

NET_PARA_REPEATERS_MIN_POWER *146*
Set minimum required received power at a repeater to consider it in network planning.

NET_PARA_POWER_SUPERPOSITION *147*
Set minimum required received power at a repeater to consider it in network planning.

NET_PARA_ANTENNA_MASKING_MODE *121*
Activate/deactivate antenna masking for network planning results (propagation results with omni-pattern).

NET_PARA_EMC_DATABASE *122*
Set filename of the EMC specifications database for EMC/EMF Analysis.

NET_PARA_COMPONENTS_DATABASE *143*
Save components catalogue database name.

Parameters for Time-Variant network projects

NET_PARA_TIME_VARIANT_MODE *123*
Set time-variant mode: 0 = Stationary simulation, 1 = Time-variant simulation with time interval, and 2 = Time-variant simulation with arbitrary time steps.

NET_PARA_TIME_VARIANT_START *124*
Set start time (s). Only relevant Time-variant simulation with time interval mode.

NET_PARA_TIME_VARIANT_END *125*
Set end time (s). Only relevant Time-variant simulation with time interval mode.

NET_PARA_TIME_VARIANT_INTERVAL *126*
Set interval time between two time steps time (s). Only relevant Time-variant simulation with time interval mode.

NET_PARA_TIME_VARIANT_ADD_STEP *127*
Add an arbitrary time step. Only relevant Time-variant simulation with arbitrary time steps.

Parameters for polarization-dependent co-channel interference that depends on the polarization and on the LOS/NLOS situation.

NET_PARA_POL_INTERFERENCE_MODE *130*
Set polarization-dependent interference mode: 0 = disabled, 2 = enabled.

NET_PARA_POL_INTERFERENCE_SAME_POL_LOS_LOS *131*
Set interference ratio If transmitting antennas have the same polarization, and transmitters (server, interferer) have LOS.

NET_PARA_POL_INTERFERENCE_SAME_POL_LOS_NLOS *132*
Set interference ratio If transmitting antennas have the same polarization, one transmitters has LOS, and the other transmitter has NLOS.

NET_PARA_POL_INTERFERENCE_SAME_POL_NLOS_NLOS *133*
Set interference ratio If transmitting antennas have the same polarization, and transmitters (server, interferer) have NLOS.

NET_PARA_POL_INTERFERENCE_SAME_POL_UNKOWN_LOS *134*
Set interference ratio If transmitting antennas have the same polarization, and LOS state for the transmitters is unknown.

NET_PARA_POL_INTERFERENCE_DIFF_POL_LOS_LOS *135*
Set interference ratio If transmitting antennas have different polarizations, and transmitters (server, interferer) have LOS.

NET_PARA_POL_INTERFERENCE_DIFF_POL_LOS_NLOS *136*
Set interference ratio If transmitting antennas have different polarizations, one transmitters has LOS, and the other transmitter has NLOS.

NET_PARA_POL_INTERFERENCE_DIFF_POL_NLOS_NLOS *137*
Set interference ratio If transmitting antennas have different polarizations, and transmitters (server, interferer) have NLOS.

NET_PARA_POL_INTERFERENCE_DIFF_POL_UNKOWN_LOS *138*
Set interference ratio If transmitting antennas have different polarizations, and LOS state for the transmitters is unknown.

NET_PARA_POL_INTERFERENCE_ORTH_POL_LOS_LOS *139*
Set interference ratio If transmitting antennas have orthogonal polarizations, and transmitters (server, interferer) have LOS.

NET_PARA_POL_INTERFERENCE_ORTH_POL_LOS_NLOS *140*
Set interference ratio If transmitting antennas have orthogonal polarizations, one transmitters has LOS, and the other transmitter has NLOS.

NET_PARA_POL_INTERFERENCE_ORTH_POL_NLOS_NLOS *141*
Set interference ratio If transmitting antennas have orthogonal polarizations, and transmitters (server, interferer) have NLOS.

NET_PARA_POL_INTERFERENCE_ORTH_POL_UNKOWN_LOS *142*
Set interference ratio If transmitting antennas have orthogonal polarizations, and LOS state for the transmitters is unknown.

Parameters to consider handover for cell assignment.

NET_PARA_HANDOVER_CELL_ASSIGNMENT *148*
Set handover for cell assignment: 0 = disabled, 1 = enabled.

NET_PARA_HANDOVER_OFFSET_DB *149*
Set handover offset in dB.

NET_PARA_HANDOVER_TTT_MS *150*
Set handover TTT (time to trigger) in ms.

Further parameters for certain network planning result types.

NET_PARA_Neighbor_Cell_Order *151*
Set maximum distance between pixels to be considered as neighbors. Relevant if NET_RESULT_NEIGHBOUR_CELLS is enabled.

NET_PARA_MIN_NUMBER_SITE *152*
Set minimum required number of sites (if less received, no output for pixel). Relevant if NET_RESULT_NR_REC_SITE is enabled.

NET_PARA_MIN_NUMBER_TRX *153*
: Set minimum required number of TRXs (if less received, no output for pixel). Relevant if NET_RESULT_NR_REC_TRX is enabled.

NET_PARA_ADDITIONAL_ASCII_BEST_SERVER *154*
: Additional ASCII output of cell assignment: 0 = no additional output, 1 = additional output of cell assignment (reports only received cells), 2 = additional output of cell assignment (reports all cells). Relevant if NET_RESULT_BEST_SERVER is enabled.

NET_PARA_ADDITIONAL_ASCII_THROUGHPUT *155*
: Additional ASCII output of maximum achievable throughput: 0 = no additional output, 1 = additional output of data streams. Relevant if NET_RESULT_MAX_DATARATE is enabled.

Optimizer parameters.

NET_OPT_ACTIVATE *1000*
: Use optimiser.

NET_OPT_PRIO_MAX *1001*
: Set priority.

NET_OPT_USE_PRIO_MAP *1011*
: Use priority map.

NET_OPT_COMBINATION_SENSITIVITY *1013*
: Use sensitivity combination.

NET_OPT_SELECTION_SENSITIVITY *1014*
: Select site sensitivity.

NET_OPT_REC_POWER_MIN *1015*
: Minimum received power setting (must not be used).

NET_OPT_SNIR_MIN *1016*
: Minimum SNIR (must not be used).

Cell Traffic setting

NET_TRAFFIC_MODE_STATIC_HOMOGENEOUS *0*
: homogeneous traffic per cell.

NET_TRAFFIC_MODE_STATIC_LOCATION_DEP *1*
: Location dependent traffic.

NET_TRAFFIC_MODE_STATIC_MONTE_CARLO *2*
: Monte-Carlo simulation.

NET_APPLICATION_AREA_MODE_METER *0*
: Application users distributed per square meter.

NET_APPLICATION_AREA_MODE_KILOMETER *1*
: Application users distributed per square kilometer.

NET_APPLICATION_TRAFFIC_MODE_ARRIVAL_RATE *0*
: Application users distributed per square meter.

NET_APPLICATION_TRAFFIC_MODE_OFFERED_TRAFFIC *1*
: Application users distributed per square kilometer.

Network application settings

NET_PARA_APPLICATION_NAME *0*
: Set network planning application name.

NET_PARA_APPLICATION_ID *1*
: Set network planning application ID.

NET_PARA_APPLICATION_PRIORITY *2*
: Set network planning application priority.

NET_PARA_APPLICATION_ACTIVITY *3*
: Set network planning application activity.

NET_PARA_APPLICATION_TRAFFIC_MODE *4*
: Set network planning application traffic mode.

NET_CELL_SELECT_UPDATE_NEW_PATH_LOSSES *0*
: Update cell list with new path loss.

NET_CELL_SELECT_UPDATE_NEW_LOAD_VALUE *1*
: Update cell list with new load value.

Parameters for carriers.

NET_PARA_CARRIER_INDEX *0*
: Set carrier index.

NET_PARA_CARRIER_FREQ_UL *1*
: Set carrier uplink frequency.

NET_PARA_CARRIER_FREQ_DL *2*
: Set carrier downlink frequency.

NET_PARA_CARRIER_NB_IN_GROUP *3*
: Set number of carriers used in a group of carrires. Only relevant if carrier represents a group of carriers.

NET_PARA_CARRIER_INDEX_IN_GROUP *4*
: Set carriers' indexes used in a group of carrires. Only relevant if carrier represents a group of carriers.

Parameters for transmission modes / services. These are set using the function WinProp_Net_TransmissionMode_Para_Set.

NET_PARA_TRANS_MODE_NAME *0*
: Set the name of the transmit mode.

NET_PARA_TRANS_MODE_BITRATE_UL *1*
: Set the transmit mode's uplink bit rate.

NET_PARA_TRANS_MODE_BITRATE_DL *2*
: Set the transmit mode's downlink bit rate.

NET_PARA_TRANS_MODE_MIN_POWER_UL *3*
: Set the minimum transmit mode uplink power.

NET_PARA_TRANS_MODE_MIN_POWER_DL *4*
: Set the minimum transmit mode downlink power.

NET_PARA_TRANS_MODE_SNIR_UL *5*
: Set the transmit mode's uplink SNIR.

NET_PARA_TRANS_MODE_SNIR_DL *6*
: Set the transmit mode's dwonlink SNIR.

NET_PARA_TRANS_MODE_POWER_BACKOFF_UL *7*
: Set uplink power backoff.

NET_PARA_TRANS_MODE_POWER_BACKOFF_DL *8*
: Set downlink power backoff.

NET_PARA_TRANS_MODE_ACTIVITIY_UL *9*
: Set uplink activity.

NET_PARA_TRANS_MODE_ACTIVITIY_DL *10*
: Set downlink activity.

NET_PARA_TRANS_MODE_MAX_POWER_UL *11*
: Set maximum allowed uplink power.

NET_PARA_TRANS_MODE_MAX_POWER_DL *12*
: Set maximum allowed downlink power.

NET_PARA_TRANS_MODE_MIN_POWER_UL_USED *13*
: Set minimum required uplink power.

NET_PARA_TRANS_MODE_MIN_POWER_DL_USED *14*
: Set minimum required downlink power.

NET_PARA_TRANS_MODE_TIMESLOTS_UL *15*
: Set the number of timeslots (uplink).

NET_PARA_TRANS_MODE_TIMESLOTS_DL *16*
Set the number of timeslots (downlink).

NET_PARA_TRANS_MODE_SPREADING_FACTOR_UL *17*
Set the transmit mode spreading factor (uplink).

NET_PARA_TRANS_MODE_SPREADING_FACTOR_DL *18*
Set the transmit mode spreading factor (downlink).

NET_PARA_TRANS_MODE_MODULATION_UL *19*
Set modulation scheme (uplink).

NET_PARA_TRANS_MODE_MODULATION_DL *20*
Set modulation scheme (downlink).

NET_PARA_TRANS_MODE_CODERATE_K_UL *21*
Set code rate (uplink).

NET_PARA_TRANS_MODE_CODERATE_N_UL *22*
Set code rate (uplink).

NET_PARA_TRANS_MODE_CODERATE_K_DL *23*
Set code rate (downlink).

NET_PARA_TRANS_MODE_CODERATE_N_DL *24*
Set code rate (downlink).

NET_PARA_TRANS_MODE_RESOURCE_BLOCKS_UL *25*
Set number of resource blocks (uplink).

NET_PARA_TRANS_MODE_RESOURCE_BLOCKS_DL *26*
Set number of resource blocks (downlink).

NET_PARA_TRANS_MODE_OVERHEAD_RATIO_UL *27*
Set overhead ratio (uplink).

NET_PARA_TRANS_MODE_OVERHEAD_RATIO_DL *28*
Set overhead ratio (downlink).

NET_PARA_TRANS_MODE_NBR_PARALLEL_CODES_UL *29*
Set parallel used codes (uplink).

NET_PARA_TRANS_MODE_NBR_PARALLEL_CODES_DL *30*
Set parallel used codes (downlink).

NET_PARA_TRANS_MODE_DIRECTION *31*
Set transmission mode direction. Only uplink/downlink or bi-directional.

NET_PARA_TRANS_MODE_POSITION *32*
Set position.

NET_PARA_5G_NUMEROLOGY0_5MHZ_FR1 *0x01*
5 MHz bandwidth, numerology 0 in frequency range 1.

NET_PARA_5G_NUMEROLOGY1_5MHZ_FR1 *0x11*
5 MHz bandwidth, numerology 1 in frequency range 1.

NET_PARA_5G_NUMEROLOGY0_10MHZ_FR1 *0x02*
10 MHz bandwidth, numerology 0 in frequency range 1.

NET_PARA_5G_NUMEROLOGY1_10MHZ_FR1 *0x12*
10 MHz bandwidth, numerology 1 in frequency range 1.

NET_PARA_5G_NUMEROLOGY2_10MHZ_FR1 *0x22*
10 MHz bandwidth, numerology 2 in frequency range 1.

NET_PARA_5G_NUMEROLOGY0_15MHZ_FR1 *0x03*
15 MHz bandwidth, numerology 0 in frequency range 1.

NET_PARA_5G_NUMEROLOGY1_15MHZ_FR1 *0x13*
15 MHz bandwidth, numerology 1 in frequency range 1.

NET_PARA_5G_NUMEROLOGY2_15MHZ_FR1 *0x23*
15 MHz bandwidth, numerology 2 in frequency range 1.

NET_PARA_5G_NUMEROLOGY0_20MHZ_FR1 *0x04*
20 MHz bandwidth, numerology 0 in frequency range 1.

NET_PARA_5G_NUMEROLOGY1_20MHZ_FR1 *0x14*
20 MHz bandwidth, numerology 1 in frequency range 1.

NET_PARA_5G_NUMEROLOGY2_20MHZ_FR1 *0x24*
20 MHz bandwidth, numerology 2 in frequency range 1.

NET_PARA_5G_NUMEROLOGY0_25MHZ_FR1 *0x05*
25 MHz bandwidth, numerology 0 in frequency range 1.

NET_PARA_5G_NUMEROLOGY1_25MHZ_FR1 *0x15*
25 MHz bandwidth, numerology 1 in frequency range 1.

NET_PARA_5G_NUMEROLOGY2_25MHZ_FR1 *0x25*
25 MHz bandwidth, numerology 2 in frequency range 1.

NET_PARA_5G_NUMEROLOGY0_30MHZ_FR1 *0x06*
30 MHz bandwidth, numerology 0 in frequency range 1.

NET_PARA_5G_NUMEROLOGY1_30MHZ_FR1 *0x16*
30 MHz bandwidth, numerology 1 in frequency range 1.

NET_PARA_5G_NUMEROLOGY2_30MHZ_FR1 *0x26*
30 MHz bandwidth, numerology 2 in frequency range 1.

NET_PARA_5G_NUMEROLOGY0_40MHZ_FR1 *0x07*
40 MHz bandwidth, numerology 0 in frequency range 1.

NET_PARA_5G_NUMEROLOGY1_40MHZ_FR1 *0x17*
40 MHz bandwidth, numerology 1 in frequency range 1.

NET_PARA_5G_NUMEROLOGY2_40MHZ_FR1 *0x27*
40 MHz bandwidth, numerology 2 in frequency range 1.

NET_PARA_5G_NUMEROLOGY0_50MHZ_FR1 *0x08*
50 MHz bandwidth, numerology 0 in frequency range 1.

NET_PARA_5G_NUMEROLOGY1_50MHZ_FR1 *0x18*
50 MHz bandwidth, numerology 1 in frequency range 1.

NET_PARA_5G_NUMEROLOGY2_50MHZ_FR1 *0x28*
50 MHz bandwidth, numerology 2 in frequency range 1.

NET_PARA_5G_NUMEROLOGY1_60MHZ_FR1 *0x19*
60 MHz bandwidth, numerology 1 in frequency range 1.

NET_PARA_5G_NUMEROLOGY2_60MHZ_FR1 *0x29*
60 MHz bandwidth, numerology 2 in frequency range 1.

NET_PARA_5G_NUMEROLOGY1_70MHZ_FR1 *0x1A*
70 MHz bandwidth, numerology 1 in frequency range 1.

NET_PARA_5G_NUMEROLOGY2_70MHZ_FR1 *0x2A*
70 MHz bandwidth, numerology 2 in frequency range 1.

NET_PARA_5G_NUMEROLOGY1_80MHZ_FR1 *0x1B*
80 MHz bandwidth, numerology 1 in frequency range 1.

NET_PARA_5G_NUMEROLOGY2_80MHZ_FR1 *0x2B*
80 MHz bandwidth, numerology 2 in frequency range 1.

NET_PARA_5G_NUMEROLOGY1_90MHZ_FR1 *0x1C*
90 MHz bandwidth, numerology 1 in frequency range 1.

NET_PARA_5G_NUMEROLOGY2_90MHZ_FR1 *0x2C*
90 MHz bandwidth, numerology 2 in frequency range 1.

NET_PARA_5G_NUMEROLOGY1_100MHZ_FR1 *0x1D*
100 MHz bandwidth, numerology 1 in frequency range 1.

NET_PARA_5G_NUMEROLOGY2_100MHZ_FR1 *0x2D*
100 MHz bandwidth, numerology 2 in frequency range 1.

NET_PARA_5G_NUMEROLOGY2_50MHZ_FR2 *0xA8*
50 MHz bandwidth, numerology 2 in frequency range 2.

NET_PARA_5G_NUMEROLOGY3_50MHZ_FR2 *0xB8*
50 MHz bandwidth, numerology 3 in frequency range 2.

NET_PARA_5G_NUMEROLOGY2_100MHZ_FR2 *0xAD*
100 MHz bandwidth, numerology 2 in frequency range 2.

NET_PARA_5G_NUMEROLOGY3_100MHZ_FR2 *0xBD*
100 MHz bandwidth, numerology 3 in frequency range 2.

NET_PARA_5G_NUMEROLOGY2_200MHZ_FR2 *0xAE*
200 MHz bandwidth, numerology 2 in frequency range 2.

NET_PARA_5G_NUMEROLOGY3_200MHZ_FR2 *0xBE*
200 MHz bandwidth, numerology 3 in frequency range 2.

NET_PARA_5G_NUMEROLOGY3_400MHZ_FR2 *0xBF*
400 MHz bandwidth, numerology 3 in frequency range 2.

Other defines

NET_PARA_CHANNEL_NAME *0*

NET_PARA_CHANNEL_SPREADING_FACTOR *1*

NET_PARA_CHANNEL_MIN_C_TO_I *2*

Error codes for network planning.

NET_ERROR_TECHNOLOGY_NOT_AVAILABLE *1*
Network planning technologies not available.

NET_ERROR_POINTER_NOT_DEFINED *2*
Issued when a nullptr is returned during initialisation of air interfaces.

NET_ERROR_PARA_NOT_AVAILABLE *3*
Unsupported parameter passed when initialising an air interface.

The documentation was generated from the following file:

- `source/Public/AirInterfaces/AI_Define.h`

2.6 Examples

This chapter contains the following:

1. Database Conversion
2. Database Preprocessing for Indoor IRT
3. Database Preprocessing for Urban IRT
4. Database Preprocessing for Urban IRT (CNP scenario)
5. Outdoor Propagation
6. Multi-threaded Outdoor Propagation
7. Mobile Station Post-processing (RunMS) in Urban Scenarios
8. Indoor Propagation
9. Indoor Propagation in Point Mode
10. Time-variant Indoor Propagation
11. Network Planning
12. Multi-threaded Network Planning
13. Monte Carlo analysis for 5G
14. Network planning with 5G TDD
15. Propagation in Rural Scenarios
16. FMCW Radar Postprocessing in Time-variant Indoor Scenarios

2.6.1 Database Conversion

Detailed Description
This is an example of how to convert an ASCII database into the WinProp binary format. The full example is distributed with the installation.

```
#include <stdio.h>
#include <string>
#include <iostream>

#include "convert_database.h"

#ifndef API_DATA_FOLDER
#define API_DATA_FOLDER "../../api/winprop/data/"
#endif // !API_DATA_FOLDER

int main(int argc, char** argv)
{
 /* Define callback functions. */
    WinProp_Callback Callback;
    WinProp_Structure_Init_Callback(&Callback);

 Callback.Percentage = CallbackProgress;
 Callback.Message = CallbackMessage;
```

```
 Callback.Error = CallbackError;

 /* Set correct parameters for conversion. */
 WinProp_Converter WinPropConverter;
 WinProp_Structure_Init_Converter(&WinPropConverter);
 WinPropConverter.databaseNameSource = API_DATA_FOLDER "outdoor/City.oda"; // with
 file extension
 WinPropConverter.databaseNameDest = API_DATA_FOLDER "outdoor/City_odb";
 WinPropConverter.measurementUnit = "Meter";
 WinPropConverter.ConverterID = 201; // convert oda to odb
 WinPropConverter.SaveAsciiFormat = 0; // Save output database in binary format

 // Start conversion of data now: my_oda_Database.oda will be converted and saved as
 // output_Database_Path/my_oda_Database.odb.
 if (WinProp_Convert(&WinPropConverter, &Callback) != 0)
  std::cout << "Error during conversion" << std::endl;

 return 0;
}

int _STD_CALL CallbackMessage(const char * Text)
{
 if (Text == nullptr)
  return 0;

 std::cout << "\n" << Text;

 return(0);
}

int _STD_CALL CallbackError(const char * Text, int Error)
{
 if (Text == nullptr)
  return 0;

 std::cout << "\n";

#ifdef __LINUX
 std::cout << "\033[31m" << "Error (" << Error << "): "; // highlight error in red
 color
#else
 HANDLE hConsole = GetStdHandle(STD_OUTPUT_HANDLE);
 SetConsoleTextAttribute(hConsole, FOREGROUND_RED);
 std::cout << "Error (" << Error << "): ";
#endif // __LINUX
 std::cout << Text;

#ifdef __LINUX
 std::cout << "\033[0m"; // highlight error in red color
#else
 SetConsoleTextAttribute(hConsole, FOREGROUND_RED | FOREGROUND_BLUE |
 FOREGROUND_GREEN);
#endif // __LINUX

 return 0;
}

int _STD_CALL CallbackProgress(int value, const char* text)
{
 char Line[200];

 sprintf(Line, "\n%d%% %s", value, text);
```

```
 std::cout << Line;

 return(0);
}
```

2.6.2 Database Preprocessing for Indoor IRT

Detailed Description

This is an example of how to use the WinProp API to preprocess an indoor database for IRT. The full example is distributed with the installation.

```
#include <stdio.h>
#include <string>
#include <iostream>

#include "IRT_preprocess.h"

#ifndef API_DATA_FOLDER
#define API_DATA_FOLDER "../../api/winprop/data/"
#endif // !API_DATA_FOLDER

int main(int argc, char** argv)
{
 int Error = 0;
 WinProp_PrePro      PreproPara;
 WinProp_Callback    Callback;

    /* ---------------------------- Initialisations -------------------------- */
    WinProp_Structure_Init_Callback(&Callback);
 WinProp_Structure_Init_PrePro(&PreproPara);
 Callback.Percentage = CallbackProgress;
 Callback.Message = CallbackMessage;
 Callback.Error = CallbackError;

 /* ----------------- Definition of parameters for preprocessing ---------- */
 PreproPara.NrHeights = 1;
 PreproPara.Heights = (double*)malloc(sizeof(double) * PreproPara.NrHeights);
 PreproPara.Heights[0] = 1.5;
 PreproPara.Resolution = 2.0;
 PreproPara.AdaptiveResolution = 1;
 PreproPara.AdaptiveResolutionFactor = 2;
 PreproPara.MultipleInteractions = 0;
 PreproPara.ExcludeDiffractions = 1;
 PreproPara.OnlyPixelsInsideBuilding = 1;
 PreproPara.TileLength = 10.0;
 PreproPara.SegmentLength = 10.0;
 PreproPara.TimeVariantMode = 1;
 PreproPara.TimeVariantStart = 0.0;
 PreproPara.TimeVariantStop = 2.0;
 PreproPara.TimeVariantInterval = 0.5;
 PreproPara.TimeVariantNrSteps = 2;
 PreproPara.TimeVariantSteps = (double*)malloc(sizeof(double) *
 PreproPara.TimeVariantNrSteps);
 PreproPara.TimeVariantSteps[0] = 1.0;
 PreproPara.TimeVariantSteps[1] = 2.0;
```

```
 Error = WinProp_PreProcessIndoor(
  // specify full path with extension
  API_DATA_FOLDER  "/indoor/IndoorVectordatabase.ida",
  // no extension here
  API_DATA_FOLDER "/indoor/IRT_Preprocessed_database",
  &PreproPara,
  &Callback
 );


 return Error;
}

int _STD_CALL CallbackMessage(const char * Text)
{
 if (Text == nullptr)
  return 0;

 std::cout << "\n" << Text;

 return(0);
}

int _STD_CALL CallbackError(const char * Text, int Error)
{
 if (Text == nullptr)
  return 0;

 std::cout << "\n";

#ifdef __LINUX
 std::cout << "\033[31m" << "Error (" << Error << "): "; // highlight error in red
 color
#else
 HANDLE hConsole = GetStdHandle(STD_OUTPUT_HANDLE);
 SetConsoleTextAttribute(hConsole, FOREGROUND_RED);
 std::cout << "Error (" << Error << "): ";
#endif // __LINUX
 std::cout << Text;

#ifdef __LINUX
 std::cout << "\033[0m"; // highlight error in red color
#else
 SetConsoleTextAttribute(hConsole, FOREGROUND_RED | FOREGROUND_BLUE |
 FOREGROUND_GREEN);
#endif // __LINUX

 return 0;
}

int _STD_CALL CallbackProgress(int value, const char* text)
{
 char Line[200];

 sprintf(Line, "\n%d%% %s", value, text);
 std::cout << Line;

 return(0);
}
```

2.6.3 Database Preprocessing for Urban IRT

Detailed Description

This is an example of how to use the WinProp API to preprocess an urban database for IRT. The full example is distributed with the installation.

```
#include <stdio.h>
#include <string>
#include <iostream>

#include "IRT_preprocess_urban.h"

#ifndef API_DATA_FOLDER
#define API_DATA_FOLDER "../../api/winprop/data/"
#endif // !API_DATA_FOLDER

int main(int argc, char** argv)
{
 int Error = 0;
 WinProp_PreProUrban PreproPara;
 WinProp_Callback    Callback;

    // Initializations
    WinProp_Structure_Init_Callback(&Callback);
 WinProp_Structure_Init_PreProUrban(&PreproPara);
 Callback.Percentage = CallbackProgress;
 Callback.Message = CallbackMessage;
 Callback.Error = CallbackError;

 // Definition of parameters for preprocessing
 sprintf(PreproPara.FilenameBuildings, "%s", API_DATA_FOLDER  "/outdoor/City");
 sprintf(PreproPara.FilenameTopography, "%s", API_DATA_FOLDER  "/outdoor/Topo.tdb");

 PreproPara.Model = PREDMODEL_IRT;
 PreproPara.Mode = PP_MODE_IRT_3D;
 PreproPara.NrHeights = 1;
 PreproPara.Heights = (double*)malloc(sizeof(double) * (PreproPara.NrHeights));
 PreproPara.Heights[0] = 1.5;
 PreproPara.HeightAbsolute = 0;
 PreproPara.ConsiderIndoorPixels = 0;
 PreproPara.AdaptiveResolution = 0;
 PreproPara.Resolution = (double)10.;
 PreproPara.SegmentHorizontal = (double)100.;
 PreproPara.SegmentVertical = (double)100.;
 PreproPara.TileHorizontal = (double)100.;
 PreproPara.TileVertical = (double)100.;
 PreproPara.ConsiderTopography = 1;
 PreproPara.AbsoluteBuildingHeights = 0;
 PreproPara.MultipleInteractions = 1;

 PreproPara.SphericMode = PP_MODE_IRT_ZONE_OFF;

 // Enable multi threading
 PreproPara.MultiThreading = 2;

 // Set IRT polygon
 PreproPara.CornersPolygon = (COORDPOINT*)malloc(sizeof(COORDPOINT) * 8);
```

```
 if (PreproPara.CornersPolygon != nullptr)
    {
        PreproPara.NrCornersPolygon = 8;
  PreproPara.CornersPolygon[0] = { 324.424, 2551.730, PreproPara.Heights[0] };
  PreproPara.CornersPolygon[1] = { 803.687, 2971.085, PreproPara.Heights[0] };
  PreproPara.CornersPolygon[2] = { 1660.829, 2961.869, PreproPara.Heights[0] };
  PreproPara.CornersPolygon[3] = { 2084.793, 2445.740, PreproPara.Heights[0] };
  PreproPara.CornersPolygon[4] = { 2084.793, 1501.039, PreproPara.Heights[0] };
  PreproPara.CornersPolygon[5] = { 1582.488, 1127.767, PreproPara.Heights[0] };
  PreproPara.CornersPolygon[6] = { 817.512, 1141.592, PreproPara.Heights[0] };
  PreproPara.CornersPolygon[7] = { 333.641, 1514.864, PreproPara.Heights[0] };
 }

 // Compute the preprocessing
 Error = OutdoorPlugIn_ComputePrePro(
  &PreproPara,
  API_DATA_FOLDER "/outdoor/City_IRT_API", // no extension here
  nullptr,
  nullptr,
  &Callback,
  nullptr
 );

 if (PreproPara.Heights != NULL)
  free(PreproPara.Heights);
 PreproPara.Heights = nullptr;
 PreproPara.NrHeights = 0;
 if (PreproPara.CornersPolygon != nullptr)
  free(PreproPara.CornersPolygon);

 return Error;
}

int _STD_CALL CallbackMessage(const char * Text)
{
 if (Text == nullptr)
  return 0;

 std::cout << "\n" << Text;

 return(0);
}

int _STD_CALL CallbackError(const char * Text, int Error)
{
 if (Text == nullptr)
  return 0;

 std::cout << "\n";

#ifdef __LINUX
 std::cout << "\033[31m" << "Error (" << Error << "): "; // highlight error in red
 color
#else
 HANDLE hConsole = GetStdHandle(STD_OUTPUT_HANDLE);
 SetConsoleTextAttribute(hConsole, FOREGROUND_RED);
 std::cout << "Error (" << Error << "): ";
#endif // __LINUX
 std::cout << Text;

#ifdef __LINUX
 std::cout << "\033[0m"; // highlight error in red color
```

```
#else
 SetConsoleTextAttribute(hConsole, FOREGROUND_RED | FOREGROUND_BLUE |
 FOREGROUND_GREEN);
#endif // __LINUX

 return 0;
}

int _STD_CALL CallbackProgress(int value, const char* text)
{
 char Line[200];

 sprintf(Line, "\n%d%% %s", value, text);
 std::cout << Line;

 return(0);
}
```

2.6.4 Database Preprocessing for Urban IRT (CNP scenario)

Detailed Description

This is an example of how to use the WinProp API to preprocess a mixed urban-indoor database (CNP scenario) for IRT. The full example is distributed with the installation.

```
#include <stdio.h>
#include <string>
#include <iostream>

#include "IRT_preprocess_urban_CNP.h"

#ifndef API_DATA_FOLDER
#define API_DATA_FOLDER "../../api/winprop/data/"
#endif // !API_DATA_FOLDER

int main(int argc, char** argv)
{
 int Error = 0;
 WinProp_PreProUrban PreproPara;
 WinProp_Callback    Callback;

    // Initializations
    WinProp_Structure_Init_Callback(&Callback);
 WinProp_Structure_Init_PreProUrban(&PreproPara);
 Callback.Percentage = CallbackProgress;
 Callback.Message = CallbackMessage;
 Callback.Error = CallbackError;

 // Definition of parameters for preprocessing
 sprintf(PreproPara.FilenameBuildings, "%s", API_DATA_FOLDER  "/outdoor/
OfficeParkCNP");

 PreproPara.Model = PREDMODEL_IRT;
 PreproPara.Mode = PP_MODE_IRT_3D;
```

```
 PreproPara.NrHeights = 1;
 PreproPara.Heights = (double*)malloc(sizeof(double) * (PreproPara.NrHeights));
 PreproPara.Heights[0] = 1.5;
 PreproPara.HeightAbsolute = 0;
 PreproPara.ConsiderIndoorPixels = 1;
 PreproPara.AdaptiveResolution = 0;
 PreproPara.Resolution = (double)10.;
 PreproPara.SegmentHorizontal = (double)100.;
 PreproPara.SegmentVertical = (double)100.;
 PreproPara.TileHorizontal = (double)100.;
 PreproPara.TileVertical = (double)100.;
 PreproPara.ConsiderTopography = 0;
 PreproPara.AbsoluteBuildingHeights = 0;
 PreproPara.MultipleInteractions = 1;

 PreproPara.SphericMode = PP_MODE_IRT_ZONE_OFF;

 // Enable multi threading
 PreproPara.MultiThreading = 2;

 // Set IRT preprocessing area
 PreproPara.lowerLeft = { 74.437, 32.228, PreproPara.Heights[0] };
 PreproPara.upperRight = { 184.812, 159.853, PreproPara.Heights[0] };

 // Set IRT CNP parameters
 PreproPara.CNPIndoorMode = 2;
 PreproPara.CNPIndoorTile = 5.0;
 PreproPara.CNPIndoorSegment = 5.0;
 PreproPara.CNPResolution = 2.0;
 PreproPara.CNPNrHeights = 1;
 PreproPara.CNPHeights = (double*)malloc(sizeof(double) * (PreproPara.CNPNrHeights));
 PreproPara.CNPHeights[0] = 1.5;
 PreproPara.CNPUrbanTile = 5.0f;
 PreproPara.CNPRedRes = 1;
 PreproPara.CNPRedResFactor = 2;
 PreproPara.CNPSphericZone = 0;
 PreproPara.CNPSphericRadius = 100.0;

 // Compute the preprocessing
 Error = OutdoorPlugIn_ComputePrePro(
  &PreproPara,
  API_DATA_FOLDER "/outdoor/OfficeParkCNP_IRT_API", // no extension here
  nullptr,
  nullptr,
  &Callback,
  nullptr
 );

 if (PreproPara.CornersPolygon != nullptr)
  free(PreproPara.CornersPolygon);
 if (PreproPara.Heights != NULL)
  free(PreproPara.Heights);
 if (PreproPara.CNPHeights != NULL)
  free(PreproPara.CNPHeights);
 PreproPara.Heights = nullptr;
 PreproPara.NrHeights = 0;
 PreproPara.CNPHeights = nullptr;
 PreproPara.CNPNrHeights = 0;

 return Error;
}
```

```
int _STD_CALL CallbackMessage(const char * Text)
{
 if (Text == nullptr)
  return 0;

 std::cout << "\n" << Text;

 return(0);
}

int _STD_CALL CallbackError(const char * Text, int Error)
{
 if (Text == nullptr)
  return 0;

 std::cout << "\n";

#ifdef __LINUX
 std::cout << "\033[31m" << "Error (" << Error << "): "; // highlight error in red
 color
#else
 HANDLE hConsole = GetStdHandle(STD_OUTPUT_HANDLE);
 SetConsoleTextAttribute(hConsole, FOREGROUND_RED);
 std::cout << "Error (" << Error << "): ";
#endif // __LINUX
 std::cout << Text;

#ifdef __LINUX
 std::cout << "\033[0m"; // highlight error in red color
#else
 SetConsoleTextAttribute(hConsole, FOREGROUND_RED | FOREGROUND_BLUE |
 FOREGROUND_GREEN);
#endif // __LINUX

 return 0;
}

int _STD_CALL CallbackProgress(int value, const char* text)
{
 char Line[200];

 sprintf(Line, "\n%d%% %s", value, text);
 std::cout << Line;

 return(0);
}
```

2.6.5 Outdoor Propagation

Detailed Description

This is an example of how to directly carry outdoor propagation. The full example is distributed with the installation.

```
#include <stdio.h>
```

```
#include <string>
#include <iostream>

#include "outdoor_propagation.h"

#ifndef API_DATA_FOLDER
#define API_DATA_FOLDER "../../api/winprop/data/"
#endif // !API_DATA_FOLDER

int main(int argc, char** argv)
{
 int                  Error = 0;
 WinProp_ParaMain     GeneralParameters;
 WinProp_Antenna      Antenna;
 WinProp_Callback     Callback;
 WinProp_Result       Resultmatrix;
 WinProp_RayMatrix    RayMatrix;
 WinProp_Propagation_Results OutputResults;

 // name of .opb database without extensions
 const char* my_opb_database = API_DATA_FOLDER "outdoor/City";

 /* ------------------------ Initialisation of parameters ----------------------*/
 WinProp_Structure_Init_ParameterMain(&GeneralParameters);
    WinProp_Structure_Init_Antenna(&Antenna);
    WinProp_Structure_Init_Callback(&Callback);
 WinProp_Structure_Init_Result(&Resultmatrix);
 WinProp_Structure_Init_RayMatrix(&RayMatrix);
 WinProp_Structure_Init_Propagation_Results(&OutputResults);

 /*---------------- Definition of scenario -------------------------------------*/
 /* Definition of general parameters. */
 GeneralParameters.ScenarioMode = SCENARIOMODE_URBAN; // Urban prediction
 GeneralParameters.PredictionModelUrban = PREDMODEL_UDP; // Use Dominant Path Model

 /* Definition of prediction area. */
 GeneralParameters.UrbanLowerLeftX = 100.0;
 GeneralParameters.UrbanLowerLeftY = 100.0;
 GeneralParameters.UrbanUpperRightX = 2200.0;
 GeneralParameters.UrbanUpperRightY = 3200.0;

 /* Copy coordinates to prediction area of second model (not yet used) */
 GeneralParameters.RuralLowerLeftX = GeneralParameters.UrbanLowerLeftX;
 GeneralParameters.RuralLowerLeftY = GeneralParameters.UrbanLowerLeftY;
 GeneralParameters.RuralUpperRightX = GeneralParameters.UrbanUpperRightX;
 GeneralParameters.RuralUpperRightY = GeneralParameters.UrbanUpperRightY;

 double PredictionHeight = 1.5;

 /* Size of matrix with results. */
 GeneralParameters.Resolution = 10.0;                    // Resolution in meter
 GeneralParameters.NrLayers = 1;                         // Number of prediction
 heights
 GeneralParameters.PredictionHeights = &PredictionHeight; // Prediction height in
 meter

 /* Building vector data and topography. */
 GeneralParameters.BuildingsMode = BUILDINGSMODE_UDP; // load a .opb database (urban
 + topography)
 sprintf(GeneralParameters.BuildingsName, "%s", my_opb_database); // Vector database
 in WinProp format

 /*--------------------------- Definition of antenna --------------------------*/
```

```
 /* Position and configuration. */
 Antenna.Id = 1;
 Antenna.SiteId = 1;
 Antenna.Longitude_X = 1100.0;
 Antenna.Latitude_Y = 2100.0;
 Antenna.Height = 20.0;
 Antenna.Power = 43.0; // Power in dBm
 Antenna.Frequency = 1800.0; // Frequency in MHz
 sprintf(Antenna.Name, "%s", "Antenna 1");

 /* Definition of outputs to be computed and written in WinProp format. */
 GeneralParameters.OutputResults = &OutputResults;
    sprintf(OutputResults.ResultPath, "%s", API_DATA_FOLDER "output/
outdoor_propagation"); // Output data directory
 OutputResults.FieldStrength = 1;
 OutputResults.PathLoss = 1;
 OutputResults.StatusLOS = 1;
 OutputResults.RayFilePropPaths = 1;
 OutputResults.StrFilePropPaths = 1;

 /*--------------------------- Callbacks ----------------------------------------*/
 Callback.Percentage = CallbackProgress;
 Callback.Message = CallbackMessage;
 Callback.Error = CallbackError;

 /*----------------------- Compute outdoor prediction -------------------------*/
 Error = OutdoorPlugIn_ComputePrediction(
  &Antenna, &GeneralParameters, NULL, 0, NULL, NULL, NULL, NULL, &Callback,
 &Resultmatrix, &RayMatrix, NULL, NULL);

 /*----------------------- Do something with results --------------------------*/
 if (Error == 0) {
  const char* Filename = API_DATA_FOLDER "output/outdoor_propagation/
PowerResults.txt";
  write_ascii(&Resultmatrix, Filename);
 }
 else {
  /* Error during prediction. Print error message. */
  CallbackError(GeneralParameters.ErrorMessageMain, Error);
 }

 /*----------------------------- Free memory ----------------------------------*/
 WinProp_FreeResult(&Resultmatrix);
 WinProp_FreeRayMatrix(&RayMatrix);

 return 0;
}

int _STD_CALL CallbackMessage(const char * Text)
{
 if (Text == nullptr)
  return 0;

 std::cout << "\n" << Text;

 return(0);
}

int _STD_CALL CallbackError(const char * Text, int Error)
{
 if (Text == nullptr)
  return 0;
```

```
 std::cout << "\n";

#ifdef __LINUX
 std::cout << "\033[31m" << "Error (" << Error << "): "; // highlight error in red
 color
#else
 HANDLE hConsole = GetStdHandle(STD_OUTPUT_HANDLE);
 SetConsoleTextAttribute(hConsole, FOREGROUND_RED);
 std::cout << "Error (" << Error << "): ";
#endif // __LINUX
 std::cout << Text;

#ifdef __LINUX
 std::cout << "\033[0m"; // highlight error in red color
#else
 SetConsoleTextAttribute(hConsole, FOREGROUND_RED | FOREGROUND_BLUE |
 FOREGROUND_GREEN);
#endif // __LINUX

 return 0;
}

int _STD_CALL CallbackProgress(int value, const char* text)
{
 char Line[200];

 sprintf(Line, "\n%d%% %s", value, text);
 std::cout << Line;

 return(0);
}

void write_ascii(const WinProp_Result* Resultmatrix, const char* Filename) {
 FILE* OutputFile = fopen(Filename,"w");
 if (OutputFile)
 {
  /* Loop all pixels. */
  for (int x = 0; x < Resultmatrix->Columns; x++)
  {
   for (int y = 0; y < Resultmatrix->Lines; y++)
   {
    /* Compute real coordinates. */
    double Coordinate_X = Resultmatrix->LowerLeftX + ((double)x + 0.5) *
 Resultmatrix->Resolution;
    double Coordinate_Y = Resultmatrix->LowerLeftY + ((double)y + 0.5) *
 Resultmatrix->Resolution;

    /* Check if pixel was computed or not */
    if (Resultmatrix->Matrix[0][x][y] > -1000)
     fprintf(OutputFile, "%.2f\t%.2f\t%.2f\n", Coordinate_X, Coordinate_Y,
 Resultmatrix->Matrix[0][x][y]);
   }
  }

  /* Close file. */
  fclose(OutputFile);
 }
 else
  printf("\nCould not open the File: %s for writing.\n",Filename);
}
```

2.6.6 Multi-threaded Outdoor Propagation

Detailed Description

This is an example of how to use the WinProp API for multi-threaded outdoor propagation with OpenMP. The full example is distributed with the installation.

```
#include <stdio.h>
#include <string>
#include <iostream>

#ifdef _OPENMP
#include <omp.h>
#endif

#include "outdoor_propagation_openmp.h"

#ifndef API_DATA_FOLDER
#define API_DATA_FOLDER "../../api/winprop/data/"
#endif // !API_DATA_FOLDER

int main(int argc, char** argv)
{
 // Load topography and building data once for all computations.
 TOPOGRAPHY TopographyData;
 InterfaceTopoInit(&TopographyData);
 InterfaceLoadTopoASC(&TopographyData, API_DATA_FOLDER "outdoor/Topo.asc");

 URBAN_BUILDINGS BuildingData;
 InterfaceBuildingsInit(&BuildingData);
 InterfaceLoadBuildingsASCII(&BuildingData, API_DATA_FOLDER "outdoor/City.oda");

 // Write additional results
 WinProp_Propagation_Results OutputResults;
 WinProp_Structure_Init_Propagation_Results(&OutputResults);
 sprintf(OutputResults.ResultPath, "%s", API_DATA_FOLDER "output/
outdoor_propagation_openmp"); // Output data directory
 OutputResults.FieldStrength = 1;
 OutputResults.PathLoss = 1;
 OutputResults.StatusLOS = 1;

 // Set the number of parallel threads.
#ifdef _OPENMP
 omp_set_num_threads(4);
#endif

 // As an example we're computing the same antenna location in the same environment
 with 16
 // different frequencies.
#pragma omp parallel for schedule(dynamic,1)
 for (int i = 0; i < 16; i++)
 {
  int Error = 0;

  WinProp_ParaMain GeneralParameters;
```

```
 WinProp_Structure_Init_ParameterMain(&GeneralParameters);

 // Definition of scenario
 /* Definition of general parameters. */
 GeneralParameters.ScenarioMode = SCENARIOMODE_URBAN; // Urban prediction
 GeneralParameters.PredictionModelUrban = PREDMODEL_UDP; // Use Dominant Path Model

 /* Definition of prediction area. */
 GeneralParameters.UrbanLowerLeftX = 100.0;
 GeneralParameters.UrbanLowerLeftY = 100.0;
 GeneralParameters.UrbanUpperRightX = 2200.0;
 GeneralParameters.UrbanUpperRightY = 3200.0;

 /* Copy coordinates to prediction area of second model (not yet used) */
 GeneralParameters.RuralLowerLeftX = GeneralParameters.UrbanLowerLeftX;
 GeneralParameters.RuralLowerLeftY = GeneralParameters.UrbanLowerLeftY;
 GeneralParameters.RuralUpperRightX = GeneralParameters.UrbanUpperRightX;
 GeneralParameters.RuralUpperRightY = GeneralParameters.UrbanUpperRightY;

 double PredictionHeight = 1.5;

 /* Size of matrix with results. */
 GeneralParameters.Resolution = 10.0;                        // Resolution in meter
 GeneralParameters.NrLayers = 1;                             // Number of prediction
heights
 GeneralParameters.PredictionHeights = &PredictionHeight;    // Prediction height in
meter

 /* Building vector data and topography. */
 GeneralParameters.BuildingsMode = BUILDINGSMODE_RAM;
 GeneralParameters.Buildings = BuildingData;

 /* Topography data */
 GeneralParameters.TopographyMode = TOPOMODE_RAM;
 GeneralParameters.Topography = TopographyData;

 /* Write some results to disk */
 GeneralParameters.OutputResults = &OutputResults;

 // Callbacks
       WinProp_Callback Callback;
       WinProp_Structure_Init_Callback(&Callback);
 Callback.Percentage = CallbackProgress;
 Callback.Message = CallbackMessage;
 Callback.Error = CallbackError;

 /* Position and configuration. */
 WinProp_Antenna Antenna;
 WinProp_Structure_Init_Antenna(&Antenna);
 Antenna.Id = i + 1;
 Antenna.SiteId = 1;
 Antenna.Longitude_X = 1100.0;
 Antenna.Latitude_Y = 2100.0;
 Antenna.Height = 20.0;
 Antenna.Power = 43.0; // Power in dBm
 Antenna.Frequency = 900.0 + 100. * i; // Frequency in MHz

 sprintf(Antenna.Name, "Antenna %i", i);

 WinProp_Result Resultmatrix;
 WinProp_RayMatrix RayMatrix;
 WinProp_Structure_Init_Result(&Resultmatrix);
```

```
  WinProp_Structure_Init_RayMatrix(&RayMatrix);

  // Compute outdoor prediction
  Error = OutdoorPlugIn_ComputePrediction(
   &Antenna, &GeneralParameters, NULL, 0, NULL, NULL, NULL, NULL, &Callback,
 &Resultmatrix, &RayMatrix, NULL, NULL);

  // Do something with results
  if (Error == 0)
  {
   char Filename[500];
   sprintf(Filename, API_DATA_FOLDER "output/outdoor_propagation_openmp/Results
 Antenna %i.txt", i);
   write_ascii(&Resultmatrix, Filename);
  }
  else
  {
   /* Error during prediction. Print error message. */
   CallbackError(GeneralParameters.ErrorMessageMain, Error);
  }

  // Free per thread memory
  WinProp_FreeResult(&Resultmatrix);
  WinProp_FreeRayMatrix(&RayMatrix);
 }

 // Free global memory
 InterfaceTopoFree(&TopographyData);
 InterfaceBuildingsInit(&BuildingData);

 return 0;
}

int _STD_CALL CallbackMessage(const char * Text)
{
 if (Text == nullptr)
  return 0;

 std::cout << "\n" << Text;

 return(0);
}

int _STD_CALL CallbackError(const char * Text, int Error)
{
 if (Text == nullptr)
  return 0;

 std::cout << "\n";

#ifdef __LINUX
 std::cout << "\033[31m" << "Error (" << Error << "): "; // highlight error in red
 color
#else
 HANDLE hConsole = GetStdHandle(STD_OUTPUT_HANDLE);
 SetConsoleTextAttribute(hConsole, FOREGROUND_RED);
 std::cout << "Error (" << Error << "): ";
#endif // __LINUX
 std::cout << Text;

#ifdef __LINUX
 std::cout << "\033[0m"; // highlight error in red color
```

```
#else
 SetConsoleTextAttribute(hConsole, FOREGROUND_RED | FOREGROUND_BLUE |
 FOREGROUND_GREEN);
#endif // __LINUX

 return 0;
}

int _STD_CALL CallbackProgress(int value, const char* text)
{
 char Line[200];

 sprintf(Line, "\n%d%% %s", value, text);
 std::cout << Line;

 return(0);
}

void write_ascii(const WinProp_Result* Resultmatrix, const char* Filename) {
 FILE* OutputFile = fopen(Filename,"w");
 if (OutputFile)
 {
  /* Loop all pixels. */
  for (int x = 0; x < Resultmatrix->Columns; x++)
  {
   for (int y = 0; y < Resultmatrix->Lines; y++)
   {
    /* Compute real coordinates. */
    double Coordinate_X = Resultmatrix->LowerLeftX + ((double)x + 0.5) *
 Resultmatrix->Resolution;
    double Coordinate_Y = Resultmatrix->LowerLeftY + ((double)y + 0.5) *
 Resultmatrix->Resolution;

    /* Check if pixel was computed or not */
    if (Resultmatrix->Matrix[0][x][y] > -1000)
     fprintf(OutputFile, "%.2f\t%.2f\t%.2f\n", Coordinate_X, Coordinate_Y,
 Resultmatrix->Matrix[0][x][y]);
   }
  }

  /* Close file. */
  fclose(OutputFile);
 }
 else
  printf("\nCould not open the File: %s for writing.\n",Filename);
}
```

2.6.7 Mobile Station Post-processing (RunMS) in Urban Scenarios

Detailed Description

This is an example of how to carry out mobile station post-processing, with the WinProp API, in an urban scenario. The full example is distributed with the installation.

```
#include <stdio.h>
#include <string>
#include <iostream>

#include "outdoor_propagation_runms.h"

#ifndef API_DATA_FOLDER
#define API_DATA_FOLDER "../../api/winprop/data/"
#endif // !API_DATA_FOLDER

int main(int argc, char** argv)
{
 int                  Error = 0;
 WinProp_ParaMain     GeneralParameters;
 WinProp_Antenna      Antenna;
 WinProp_Callback     Callback;
 WinProp_ResultTrajectoryList WinPropResultTrajectories;
 WinProp_Propagation_Results OutputResults;

 // Set up input trajectory
 int NrWinPropTrajectories = 1;
 WinProp_Trajectory* WinPropTrajectories = nullptr;
 WinPropTrajectories = (WinProp_Trajectory*)malloc(sizeof(WinProp_Trajectory) *
 NrWinPropTrajectories);
 if (WinPropTrajectories != nullptr)
 {
  WinProp_Structure_Init_Trajectory(&WinPropTrajectories[0]);
  WinPropTrajectories[0].Id = 1;
  WinPropTrajectories[0].PointSize = 5.0;
  WinPropTrajectories[0].samplingResolution = 1.0;
  WinPropTrajectories[0].NrPoints = 9;
  WinPropTrajectories[0].Points =
 (WinProp_Trajectory_Point*)malloc(sizeof(WinProp_Trajectory_Point) *
 WinPropTrajectories[0].NrPoints);
  if (WinPropTrajectories[0].Points != nullptr)
  {
   for (int i = 0; i < WinPropTrajectories[0].NrPoints; i++)
    WinProp_Structure_Init_TrajectoryPoint(&WinPropTrajectories[0].Points[i]);

   WinPropTrajectories[0].Points[0].location.x = 476101.49;
   WinPropTrajectories[0].Points[0].location.y = 552413.58;
   WinPropTrajectories[0].Points[0].location.z = 1.5;

   WinPropTrajectories[0].Points[1].location.x = 476597.37;
   WinPropTrajectories[0].Points[1].location.y = 552604.19;
   WinPropTrajectories[0].Points[1].location.z = 1.5;

   WinPropTrajectories[0].Points[2].location.x = 476607.80;
   WinPropTrajectories[0].Points[2].location.y = 552620.57;
   WinPropTrajectories[0].Points[2].location.z = 1.5;
```

```
    WinPropTrajectories[0].Points[3].location.x = 476391.13;
    WinPropTrajectories[0].Points[3].location.y = 553024.13;
    WinPropTrajectories[0].Points[3].location.z = 1.5;

    WinPropTrajectories[0].Points[4].location.x = 476333.05;
    WinPropTrajectories[0].Points[4].location.y = 553030.83;
    WinPropTrajectories[0].Points[4].location.z = 1.5;

    WinPropTrajectories[0].Points[5].location.x = 476201.26;
    WinPropTrajectories[0].Points[5].location.y = 552888.62;
    WinPropTrajectories[0].Points[5].location.z = 1.5;

    WinPropTrajectories[0].Points[6].location.x = 475830.47;
    WinPropTrajectories[0].Points[6].location.y = 552670.46;
    WinPropTrajectories[0].Points[6].location.z = 1.5;

    WinPropTrajectories[0].Points[7].location.x = 475810.36;
    WinPropTrajectories[0].Points[7].location.y = 552624.29;
    WinPropTrajectories[0].Points[7].location.z = 1.5;

    WinPropTrajectories[0].Points[8].location.x = 476082.88;
    WinPropTrajectories[0].Points[8].location.y = 552423.26;
    WinPropTrajectories[0].Points[8].location.z = 1.5;
  }
}

// name of .odb database without extension
const char* my_odb_database = API_DATA_FOLDER "outdoor/Frankfurt";

/* ------------------------ Initialisation of parameters ----------------------*/
WinProp_Structure_Init_ParameterMain(&GeneralParameters);
   WinProp_Structure_Init_Antenna(&Antenna);
   WinProp_Structure_Init_Callback(&Callback);
WinProp_Structure_Init_ResultTrajectoryList(&WinPropResultTrajectories);
WinProp_Structure_Init_Propagation_Results(&OutputResults);

/*---------------- Definition of scenario -------------------------------------*/
/* Definition of general parameters. */
GeneralParameters.ScenarioMode = SCENARIOMODE_URBAN; // Urban prediction
GeneralParameters.PredictionModelUrban = PREDMODEL_UDP; // Use Dominant Path Model

/* Definition of prediction trajectory. */
GeneralParameters.UrbanLowerLeftX = 475800.0;
GeneralParameters.UrbanLowerLeftY = 552400.0;
GeneralParameters.UrbanUpperRightX = 476620.0;
GeneralParameters.UrbanUpperRightY = 553040.0;

/* Copy coordinates to prediction area of second model (not yet used) */
GeneralParameters.RuralLowerLeftX = GeneralParameters.UrbanLowerLeftX;
GeneralParameters.RuralLowerLeftY = GeneralParameters.UrbanLowerLeftY;
GeneralParameters.RuralUpperRightX = GeneralParameters.UrbanUpperRightX;
GeneralParameters.RuralUpperRightY = GeneralParameters.UrbanUpperRightY;

double PredictionHeight = 1.5;

/* Size of matrix with results. */
GeneralParameters.Resolution = 5.0;                       // Resolution in meter
GeneralParameters.NrLayers = 1;                           // Number of prediction
heights
GeneralParameters.PredictionHeights = &PredictionHeight; // Prediction height in
meter

/* Building vector data and topography. */
```

```
 GeneralParameters.BuildingsMode = BUILDINGSMODE_BINARY; // load a .opb database
 (urban)
 sprintf(GeneralParameters.BuildingsName, "%s", my_odb_database); // Vector database
 in WinProp format

 /*--------------------------- Definition of antenna ---------------------------*/
 /* Position and configuration. */
 Antenna.Id = 1;
 Antenna.SiteId = 1;
 Antenna.Longitude_X = 476079.78;
 Antenna.Latitude_Y = 552495.05;
 Antenna.Height = 25.0;
 Antenna.Power = 40.0; // Power in dBm
    Antenna.Frequency = 2000.0; // Frequency in MHz
    sprintf(Antenna.Name, "%s", "Antenna 1");

 /* Definition of outputs to be computed and written in WinProp format. */
    GeneralParameters.OutputResults = &OutputResults;
    sprintf(OutputResults.ResultPath, "%s", API_DATA_FOLDER "output/
outdoor_propagation_runms/propagation"); // Output data directory
 OutputResults.FieldStrength = 1;
 OutputResults.PathLoss = 1;
 OutputResults.StatusLOS = 1;
 OutputResults.RayFilePropPaths = 1;
 OutputResults.StrFilePropPaths = 1;

 /*--------------------------- Callbacks ---------------------------------------*/
 Callback.Percentage = CallbackProgress;
 Callback.Message = CallbackMessage;
 Callback.Error = CallbackError;

 /*------------------------ Compute outdoor prediction ---------------------------*/
 Error = OutdoorPlugIn_ComputeTrajectories(
  &Antenna, &GeneralParameters, NULL, NULL,
  WinPropTrajectories, NrWinPropTrajectories, &Callback, &WinPropResultTrajectories);

 /*------------------------ Do something with results ----------------------------*/
 if (Error == 0)
 {
  write_ascii(&WinPropResultTrajectories, API_DATA_FOLDER "output/
outdoor_propagation_runms/propagation/OutdoorRunPRO.txt");
 }
 else {
  /* Error during prediction. Print error message. */
  CallbackError(GeneralParameters.ErrorMessageMain, Error);
 }

 /*------------------------ Rx antenna superposition (RunMS) --------------------*/
 // Create project
 WinProp_SuperposeMS superposeMS;

 /* Definition of antenna pattern. */
 WinProp_Pattern     WinPropPattern;
 WinProp_Structure_Init_Pattern(&WinPropPattern);
 WinPropPattern.Mode = PATTERN_MODE_FILE; // Load pattern from file
 sprintf(WinPropPattern.Filename, "%s", API_DATA_FOLDER "antennas/Antenna.apb"); //
 Pattern file (including extension)

 // Setup tx array
 WINPROP_MS_ARRAY_TYPE txArrayAPI = WINPROP_MS_ARRAY_SINGLE_ANTENNA;
 WINPROP_MS_POLARIZATION_TYPE pol = WINPROP_MS_POLARIZATION_VERTICAL;
 if (Error == 0)
```

```
  Error = superposeMS.setArray(false, txArrayAPI, 1, 0.f, 0.f, 0.f, 0);
 if (Error == 0)
  // Changing the azimuth from north over east in RunPRO to east over north
 orientation, 90deg
  // tilt is equal to horizontal orientation.
  Error = superposeMS.setArrayElement(
   false, 0, nullptr, nullptr,
   (float)(-Antenna.Azimuth + 90. + 360.),
   (float)(Antenna.Downtilt + 90.), pol, { 0., 0., 0. });

 // Setup rx array
 WINPROP_MS_ARRAY_TYPE rxArrayAPI = WINPROP_MS_ARRAY_SINGLE_ANTENNA;
 if (Error == 0)
  Error = superposeMS.setArray(true, rxArrayAPI,
   1   /*rxArray.nbrElements*/,
   0.f /*rxArray.arrayAzimuth*/,
   0.f /*rxArray.spacingHorizontal*/,
   0.f /*rxArray.radius*/,
   false /*rxArray.antCoupling != 0*/);

 if (Error == 0)
  // azimuth with east over north orientation, in case of trajectories 0deg means
 90deg right
  // from driving direction, using 180deg here to point the Rx antenna towards the Tx
 90deg tilt
  // is equal to horizontal orientation.
  Error = superposeMS.setArrayElement(
   true, 0, &WinPropPattern, nullptr,
   180.f/*azimuth*/, 90.f/*tilt*/,
   pol, { 0., 0., 0. }/*offset*/);

 if (Error == 0)
  Error = superposeMS.setPropagationTrajectories(
   0, Antenna, WinPropTrajectories, NrWinPropTrajectories,
 WinPropResultTrajectories);


 // Set computation parameter
 WinProp_MS_Para msPara;
 WinProp_Structure_Init_MS_Para(&msPara);
 // Enable ouputs
 WinProp_MS_AdditionalResults msAdditionalResult;
 WinProp_Structure_Init_MS_AdditionalResults(&msAdditionalResult);
 sprintf(msAdditionalResult.outputPath, "%s%s", API_DATA_FOLDER, "output/
outdoor_propagation_runms/runms");

 if (Error == 0)
  Error = superposeMS.compute(msPara, &msAdditionalResult, &Callback);

 /*---------------------------- Free memory ----------------------------------*/
 WinProp_Structure_Free_ResultTrajectoryList(&WinPropResultTrajectories);
 if (WinPropTrajectories != nullptr)
 {
  if (WinPropTrajectories->Points != nullptr)
   free(WinPropTrajectories->Points);
  free(WinPropTrajectories);
 }


 return 0;
}
```

```
int _STD_CALL CallbackMessage(const char * Text)
{
 if (Text == nullptr)
  return 0;

 std::cout << "\n" << Text;

 return(0);
}

int _STD_CALL CallbackError(const char * Text, int Error)
{
 if (Text == nullptr)
  return 0;

 std::cout << "\n";

#ifdef __LINUX
 std::cout << "\033[31m" << "Error (" << Error << "): "; // highlight error in red
 color
#else
 HANDLE hConsole = GetStdHandle(STD_OUTPUT_HANDLE);
 SetConsoleTextAttribute(hConsole, FOREGROUND_RED);
 std::cout << "Error (" << Error << "): ";
#endif // __LINUX
 std::cout << Text;

#ifdef __LINUX
 std::cout << "\033[0m"; // highlight error in red color
#else
 SetConsoleTextAttribute(hConsole, FOREGROUND_RED | FOREGROUND_BLUE |
 FOREGROUND_GREEN);
#endif // __LINUX

 return 0;
}

int _STD_CALL CallbackProgress(int value, const char* text)
{
 char Line[200];

 sprintf(Line, "\n%d%% %s", value, text);
 std::cout << Line;

 return(0);
}

void write_ascii(const WinProp_ResultTrajectoryList* Result, const char* Filename) {
 FILE* OutputFile = fopen(Filename, "w");
 if (OutputFile)
 {
  /* Loop all pixels. */
  for (int x = 0; x < Result->NrTrajectories; x++)
  {
   for (int y = 0; y < Result->trajectories[x].ResultPoints.NrResultPoints; y++)
   {
    /* Compute real coordinates. */
    double Coordinate_X = Result-
>trajectories[x].ResultPoints.ResultPoints[y].Location.x;
    double Coordinate_Y = Result-
>trajectories[x].ResultPoints.ResultPoints[y].Location.y;
```

```
    double Coordinate_Z = Result-
>trajectories[x].ResultPoints.ResultPoints[y].Location.z;
    double ResultValue = Result-
>trajectories[x].ResultPoints.ResultPoints[y].ResultValue;

    /* Check if pixel was computed or not */
    if (ResultValue > -1000)
     fprintf(OutputFile, "%.2f\t%.2f\t%.2f\t%.2f\n", Coordinate_X, Coordinate_Y,
 Coordinate_Z, ResultValue);
   }
  }

  /* Close file. */
  fclose(OutputFile);
 }
 else
  printf("\nCould not open the File: %s for writing.\n", Filename);
}
```

2.6.8 Indoor Propagation

Detailed Description

This is an example of how to use the WinProp API for indoor propagation. The full example is distributed with the installation.

```
#include <stdio.h>
#include <string>
#include <iostream>

#include "indoor_propagation.h"

#ifndef API_DATA_FOLDER
#define API_DATA_FOLDER "../../api/winprop/data/"
#endif // !API_DATA_FOLDER

int main(int argc, char** argv)
{
 int                 Error = 0, ProjectHandle = 0;
 WinProp_Scenario    WinPropScenario;
 WinProp_Callback    WinPropCallback;
 WinProp_Area        WinPropArea;
 WinProp_Antenna     WinPropAntenna;
 WinProp_Result      *PowerResult = NULL, *DelayResult = NULL, *LosResult = NULL;
 WinProp_RayMatrix   *RayMatrix = NULL;
 WinProp_Pattern     WinPropPattern;
 WinProp_Additional  WinPropMore;
 WinProp_Propagation_Results OutputResults;
 double              PredictionHeight = 1.5;


 /* ------------------------ Initialization of structures --------------------- */
    WinProp_Structure_Init_Scenario(&WinPropScenario);
    WinProp_Structure_Init_Callback(&WinPropCallback);
 WinProp_Structure_Init_Area(&WinPropArea);
```

```
 WinProp_Structure_Init_Antenna(&WinPropAntenna);
 WinProp_Structure_Init_Pattern(&WinPropPattern);
 WinProp_Structure_Init_Additional(&WinPropMore);
 WinProp_Structure_Init_Propagation_Results(&OutputResults);

 /* ---------- Load indoor vector database and initialise scenario ------------ */
 /* Assign database name. */
    WinPropScenario.Scenario = WINPROP_SCENARIO_INDOOR;
    sprintf(WinPropScenario.VectorDatabase, "%s", API_DATA_FOLDER "../data/indoor/
IndoorVectordatabase.idb");
 /* Define callback functions. */
 WinPropCallback.Percentage = CallbackProgress;
 WinPropCallback.Message = CallbackMessage;
 WinPropCallback.Error = CallbackError;


 /* Call the WinProp API to open a project and load the vector database. */
 Error = WinProp_Open(&ProjectHandle, &WinPropScenario, &WinPropCallback);
 /* ---------------------- Set up prediction -------------------------------- */
 if (Error == 0) {
  /* Definition of prediction area and resolution. */
  WinPropArea.LowerLeftX = -4.5;
  WinPropArea.LowerLeftY = -44.5;
  WinPropArea.UpperRightX = 101.5;
  WinPropArea.UpperRightY = 43.0;
  WinPropArea.Resolution = 1.0; // Resolution 1.0 meter
  WinPropArea.NrHeights = 1; // Number of prediction heights
  WinPropArea.Heights = &PredictionHeight; // Prediction height 1.5 meter

  /* Definition of antenna pattern. */
        WinPropPattern.Mode = PATTERN_MODE_FILE; // Load pattern from file
        sprintf(WinPropPattern.Filename, "%s", API_DATA_FOLDER "antennas/
Antenna.apb"); // Pattern file (including extension)

  /* Defintion of antenna properties. */
  WinPropAntenna.Enabled = ANTENNA_ENABLED;
  WinPropAntenna.Id = 1;
  WinPropAntenna.SiteId = 1;
  WinPropAntenna.Longitude_X = 60.25;
  WinPropAntenna.Latitude_Y = 6.25;
  WinPropAntenna.Height = 2.0; // Antenna height 2.0 meter
  WinPropAntenna.Model = WINPROP_MODEL_DPM; // Use the DPM
  WinPropAntenna.DataType = PROP_RESULT_POWER; // Compute received power
  WinPropAntenna.Power = 10.0; // Power in dBm
  WinPropAntenna.Frequency = 1800.0; // Frequency 1800 MHz
  WinPropAntenna.Pattern = &WinPropPattern; // Use pattern defined above
  WinPropAntenna.Azimuth = 270.0; // Antenna points in western direction
        WinPropAntenna.Downtilt = 1.0; // Downtilt of 1 degree

        sprintf(WinPropAntenna.Name, "%s", "my_antenna_name"); // name of the antenna

  /* Definition of outputs to be computed and written in WinProp format. */
        WinPropMore.OutputResults = &OutputResults;
        sprintf(OutputResults.ResultPath, "%s", API_DATA_FOLDER "output/
indoor_propagation"); // Output data directory
  OutputResults.FieldStrength = 1;
  OutputResults.PathLoss = 1;

  /* Further parameters: With filtering. */
  WinPropMore.ResultFiltering = 1;

  /* ---------------------- Start prediction ------------------------------ */
```

```
  Error = WinProp_Predict(
   ProjectHandle,
   &WinPropAntenna,
   &WinPropArea,
   nullptr,
   0,
   nullptr,
   &WinPropMore,
   &PowerResult,
   &DelayResult,
   &LosResult,
   &RayMatrix,
   nullptr,
   nullptr,
   nullptr);

  if (Error == 0){
   char PowerOutputFile[200];
   sprintf(PowerOutputFile, API_DATA_FOLDER "output/indoor_propagation/
my_antenna_name_Power.txt");
   write_ascii(PowerResult, PowerOutputFile);

   char LoSOutputFile[200];
   sprintf(LoSOutputFile, API_DATA_FOLDER "output/indoor_propagation/
my_antenna_name_LOS.txt");
   write_ascii(LosResult, LoSOutputFile);
  }
 }
 WinProp_Close(ProjectHandle);

 return 0;
}

int _STD_CALL CallbackMessage(const char * Text)
{
 if (Text == nullptr)
  return 0;

 std::cout << "\n" << Text;

 return(0);
}

int _STD_CALL CallbackError(const char * Text, int Error)
{
 if (Text == nullptr)
  return 0;

 std::cout << "\n";

#ifdef __LINUX
 std::cout << "\033[31m" << "Error (" << Error << "): "; // highlight error in red
 color
#else
 HANDLE hConsole = GetStdHandle(STD_OUTPUT_HANDLE);
 SetConsoleTextAttribute(hConsole, FOREGROUND_RED);
 std::cout << "Error (" << Error << "): ";
#endif // __LINUX
 std::cout << Text;

#ifdef __LINUX
 std::cout << "\033[0m"; // highlight error in red color
```

```
#else
 SetConsoleTextAttribute(hConsole, FOREGROUND_RED | FOREGROUND_BLUE |
 FOREGROUND_GREEN);
#endif // __LINUX

 return 0;
}

int _STD_CALL CallbackProgress(int value, const char* text)
{
 char Line[200];

 sprintf(Line, "\n%d%% %s", value, text);
 std::cout << Line;

 return(0);
}

// Helper functions
void write_ascii(const WinProp_Result* Resultmatrix, const char* Filename)
{
 FILE* OutputFile = fopen(Filename,"w");
 if (OutputFile)
 {
  /* Loop through WinPropall pixels. */
  for (int x = 0; x < Resultmatrix->Columns; x++)
  {
   for (int y = 0; y < Resultmatrix->Lines; y++)
   {
    /* Compute real coordinates. */
    double Coordinate_X = Resultmatrix->LowerLeftX + ((double)x + 0.5) *
 Resultmatrix->Resolution;
    double Coordinate_Y = Resultmatrix->LowerLeftY + ((double)y + 0.5) *
 Resultmatrix->Resolution;

    /* Check if pixel was computed or not */
    if (Resultmatrix->Matrix[0][x][y] > -1000)
     fprintf(OutputFile, "%.2f\t%.2f\t%.2f\n", Coordinate_X, Coordinate_Y,
 Resultmatrix->Matrix[0][x][y]);
   }
  }

  /* Close file. */
  fclose(OutputFile);
 }
 else
  printf("\nCould not open the File: %s for writing.\n",Filename);
}
```

2.6.9 Indoor Propagation in Point Mode

Detailed Description

This is an example of how to use the WinProp API for indoor propagation in point mode. The full example is distributed with the installation.

```
#include <stdio.h>
#include <string>
#include <iostream>

#include "indoor_propagation_points.h"

#ifndef API_DATA_FOLDER
#define API_DATA_FOLDER "../../api/winprop/data/"
#endif // !API_DATA_FOLDER

int main(int argc, char** argv)
{
 const int NrPoints = 3;
 int                 Error = 0, ProjectHandle = 0;
 WinProp_Scenario    WinPropScenario;
 WinProp_Callback    WinPropCallback;
 WinProp_Receiver    WinPropReceiver[NrPoints];
 WinProp_Antenna     WinPropAntenna;
 WinProp_Pattern     WinPropPattern;
 WinProp_Additional  WinPropMore;
 WinProp_Propagation_Results OutputResults;


 /* ------------------ Initialization of structures --------------------------- */
    WinProp_Structure_Init_Scenario(&WinPropScenario);
    WinProp_Structure_Init_Callback(&WinPropCallback);
 WinProp_Structure_Init_Antenna(&WinPropAntenna);
 WinProp_Structure_Init_Pattern(&WinPropPattern);
 WinProp_Structure_Init_Additional(&WinPropMore);
 WinProp_Structure_Init_Propagation_Results(&OutputResults);

 for (int loop = 0; loop < NrPoints; loop++)
 {
  WinProp_Structure_Init_Receiver(&WinPropReceiver[loop]);
 }

 /* ---------- Load indoor vector database and initialise scenario ------------ */
 /* Assign database name. */
 WinPropScenario.Scenario = WINPROP_SCENARIO_INDOOR;
 sprintf(WinPropScenario.VectorDatabase, "%s", API_DATA_FOLDER "../data/indoor/
IndoorVectordatabase.idb");

 /* Define callback functions. */
 WinPropCallback.Percentage = CallbackProgress;
 WinPropCallback.Message = CallbackMessage;
 WinPropCallback.Error = CallbackError;


 /* Call the WinProp API to open a project and load the vector database. */
 Error = WinProp_Open(&ProjectHandle, &WinPropScenario, &WinPropCallback);
 /* ---------------------- Set up prediction ------------------------------- */
 if (Error == 0) {
  /* Definition of prediction points. */
```

```
  WinPropReceiver[0].Location.x = 45.0;
  WinPropReceiver[0].Location.y = 12.0;
  WinPropReceiver[0].Location.z = 0.5;

  WinPropReceiver[1].Location.x = 70.0;
  WinPropReceiver[1].Location.y = -2.0;
  WinPropReceiver[1].Location.z = 1.5;

  WinPropReceiver[2].Location.x = 60.0;
  WinPropReceiver[2].Location.y = 2.0;
  WinPropReceiver[2].Location.z = 2.5;

  /* Definition of antenna pattern. */
  WinPropPattern.Mode = PATTERN_MODE_FILE; // Load pattern from file
  sprintf(WinPropPattern.Filename, "%s", API_DATA_FOLDER "antennas/Antenna.apb"); //
 Pattern file (including extension)

  /* Defintion of antenna properties. */
  WinPropAntenna.Enabled = ANTENNA_ENABLED;
  WinPropAntenna.Id = 1;
  WinPropAntenna.SiteId = 1;
  WinPropAntenna.Longitude_X = 60.25;
  WinPropAntenna.Latitude_Y = 6.25;
  WinPropAntenna.Height = 2.0; // Antenna height 2.0 meter
  WinPropAntenna.Model = WINPROP_MODEL_SRT; // Use the DPM
  WinPropAntenna.DataType = PROP_RESULT_POWER; // Compute received power
  WinPropAntenna.Power = 10.0; // Power in dBm
  WinPropAntenna.Frequency = 1800.0; // Frequency 1800 MHz
  WinPropAntenna.Pattern = &WinPropPattern; // Use pattern defined above
  WinPropAntenna.Azimuth = 270.0; // Antenna points in western direction
  WinPropAntenna.Downtilt = 1.0; // Downtilt of 1 degree

  sprintf(WinPropAntenna.Name, "%s", "my_antenna_name"); // name of the antenna

  /* Definition of outputs to be computed and written in WinProp format. */
  WinPropMore.OutputResults = &OutputResults;
  sprintf(OutputResults.ResultPath, "%s", API_DATA_FOLDER "output/
indoor_propagation_points/IndoorPoints_Output"); // Output data directory
  OutputResults.FieldStrength = 1;
  OutputResults.PathLoss = 1;
  OutputResults.Delay = 1;
  OutputResults.DelaySpread = 1;
  OutputResults.AngularSpreadMS = 1;
  OutputResults.AngularSpreadBS = 1;
  OutputResults.AngularMeans = 1;
  OutputResults.StatusLOS = 1;
  OutputResults.RayFilePropPaths = 1;
  OutputResults.StrFilePropPaths = 1;
  // OutputResults.StrFileTransMatrix = 1; // only possible for
 WINPROP_INTERACTION_MODEL_GTDUTD
  // WinPropMore.InteractionModel = WINPROP_INTERACTION_MODEL_GTDUTD;

  /* Further parameters: With filtering. */
  WinPropMore.ResultFiltering = 1;

        /* ----------------------- Start prediction ----------------------------- */
        WinProp_ResultPointsList* PowerResult = nullptr;
  Error = WinProp_Predict_Points(
   ProjectHandle, &WinPropAntenna, WinPropReceiver, NrPoints, nullptr, &WinPropMore,
 &PowerResult, nullptr, nullptr);

  // Write results to ascii files
  if (Error == 0)
```

```
  {
   char PowerOutputFile[200];
   sprintf(PowerOutputFile, API_DATA_FOLDER "output/indoor_propagation_points/
%s_Power.txt", WinPropAntenna.Name);
   write_ascii(PowerResult, PowerOutputFile);
  }

  /* Free memory */
     WinProp_Structure_Free_ResultPointsList(PowerResult);
 }

 WinProp_Close(ProjectHandle);

 return 0;
}

int _STD_CALL CallbackMessage(const char * Text)
{
 if (Text == nullptr)
  return 0;

 std::cout << "\n" << Text;

 return(0);
}

int _STD_CALL CallbackError(const char * Text, int Error)
{
 if (Text == nullptr)
  return 0;

 std::cout << "\n";

#ifdef __LINUX
 std::cout << "\033[31m" << "Error (" << Error << "): "; // highlight error in red
 color
#else
 HANDLE hConsole = GetStdHandle(STD_OUTPUT_HANDLE);
 SetConsoleTextAttribute(hConsole, FOREGROUND_RED);
 std::cout << "Error (" << Error << "): ";
#endif // __LINUX
 std::cout << Text;

#ifdef __LINUX
 std::cout << "\033[0m"; // highlight error in red color
#else
 SetConsoleTextAttribute(hConsole, FOREGROUND_RED | FOREGROUND_BLUE |
 FOREGROUND_GREEN);
#endif // __LINUX

 return 0;
}

int _STD_CALL CallbackProgress(int value, const char* text)
{
 char Line[200];

 sprintf(Line, "\n%d%% %s", value, text);
 std::cout << Line;

 return(0);
}
```

```
// Helper functions
void write_ascii(const WinProp_ResultPointsList * Resultmatrix, const char* Filename)
{
 FILE* OutputFile = fopen(Filename,"w");
 if (OutputFile)
 {
  for (int p = 0; p < Resultmatrix->NrResultPoints; p++)
  {
   double result = Resultmatrix->ResultPoints[p].ResultValue;
   double Coordinate_X = Resultmatrix->ResultPoints[p].Location.x;
   double Coordinate_Y = Resultmatrix->ResultPoints[p].Location.y;
   double Coordinate_Z = Resultmatrix->ResultPoints[p].Location.z;

   fprintf(OutputFile, "%.2f\t%.2f\t%.2f\t%.2f\n", Coordinate_X, Coordinate_Y,
 Coordinate_Z, result);
  }

  /* Close file. */
  fclose(OutputFile);
 }
 else
  printf("\nCould not open the File: %s for writing.\n",Filename);
}
```

2.6.10 Time-variant Indoor Propagation

Detailed Description

This is an example of how to use the WinProp API for time-variant indoor propagation. The full example is distributed with the installation.

```
#include <stdio.h>
#include <string>
#include <iostream>

#include "indoor_propagation_timevariant.h"

#ifndef API_DATA_FOLDER
#define API_DATA_FOLDER "../../api/winprop/data/"
#endif // !API_DATA_FOLDER

int main(int argc, char** argv)
{
 int                 Error = 0, ProjectHandle = 0;
 WinProp_Scenario    WinPropScenario;
 WinProp_Callback    WinPropCallback;
 WinProp_Area        WinPropArea;
 WinProp_Antenna     WinPropAntenna;
 WinProp_Dynamic_Antenna *WinPropDynAntenna;
 WinProp_Result      *PowerResult = NULL, *DelayResult = NULL, *LosResult = NULL;
 WinProp_RayMatrix   *RayMatrix = NULL;
 WinProp_Pattern     WinPropPattern;
 WinProp_Additional  WinPropMore;
 WinProp_Propagation_Results OutputResults;
```

```
 double               PredictionHeight = 1.5;


 /* ------------------ Initialization of structures -------------------------- */
    WinProp_Structure_Init_Scenario(&WinPropScenario);
    WinProp_Structure_Init_Callback(&WinPropCallback);
 WinProp_Structure_Init_Area(&WinPropArea);
 WinProp_Structure_Init_Antenna(&WinPropAntenna);
 WinProp_Structure_Init_Pattern(&WinPropPattern);
 WinProp_Structure_Init_Additional(&WinPropMore);
 WinProp_Structure_Init_Propagation_Results(&OutputResults);

 /* ---------- Load indoor vector database and initialise scenario ------------ */
 /* Assign database name. */
    WinPropScenario.Scenario = WINPROP_SCENARIO_INDOOR;
    sprintf(WinPropScenario.VectorDatabase, "%s", API_DATA_FOLDER "../data/indoor/
TestSimpleMovingBox.idb");

 /* Define callback functions. */
 WinPropCallback.Percentage = CallbackProgress;
 WinPropCallback.Message = CallbackMessage;
 WinPropCallback.Error = CallbackError;


 /* ------------------------- Set up prediction ------------------------------- */
 if (Error == 0)
 {
  /* Definition of prediction area and resolution. */
  WinPropArea.LowerLeftX = 0;
  WinPropArea.LowerLeftY = 0;
  WinPropArea.UpperRightX = 50;
  WinPropArea.UpperRightY = 30.0;
  WinPropArea.Resolution = 1.0; // Resolution 1.0 meter
  WinPropArea.NrHeights = 1; // Number of prediction heights
  WinPropArea.Heights = &PredictionHeight; // Prediction height 1.5 meter

  /* Definition of antenna pattern. */
        WinPropPattern.Mode = PATTERN_MODE_FILE; // Load pattern from file
        sprintf(WinPropPattern.Filename, "%s", API_DATA_FOLDER "antennas/
Antenna.apb"); // Pattern file (including extension)

  /* Defintion of antenna properties. */
  WinPropAntenna.Enabled = ANTENNA_ENABLED;
  WinPropAntenna.Id = 1;
  WinPropAntenna.SiteId = 1;
  WinPropAntenna.Longitude_X = 5.0;
  WinPropAntenna.Latitude_Y = 5.0;
  WinPropAntenna.Height = 2.5; // Antenna height 2.0 meter
  WinPropAntenna.Model = WINPROP_MODEL_COST231; // Use the COST MW
  WinPropAntenna.DataType = PROP_RESULT_POWER; // Compute received power
  WinPropAntenna.Power = 10.0; // Power in dBm
  WinPropAntenna.Frequency = 1800.0; // Frequency 1800 MHz
  WinPropAntenna.Pattern = &WinPropPattern; // Use pattern defined above
  WinPropAntenna.Azimuth = 270.0; // Antenna points in western direction
  WinPropAntenna.Downtilt = 1.0; // Downtilt of 1 degree

        sprintf(WinPropAntenna.Name, "%s", "my_antenna_name"); // name of the antenna

  /* Definition of outputs to be computed and written in WinProp format. */
  WinPropMore.OutputResults = &OutputResults;
  OutputResults.FieldStrength = 1;
  OutputResults.PathLoss = 1;
  OutputResults.RayFilePropPaths = 1;
```

```
  OutputResults.StrFilePropPaths = 1;

  /* Further parameters: With filtering. */
  WinPropMore.ResultFiltering = 1;

  double timeInstances[3] = { 0., .5, 1. };
  /* Call the WinProp API to open a project and load the vector database. */
  Error = WinProp_Open(&ProjectHandle, &WinPropScenario, &WinPropCallback);

  // Set time instance and output folder
  WinPropMore.TimeInstances = timeInstances;
        WinPropMore.NbrTimeInstances = 3;
        sprintf(OutputResults.ResultPath, "%s", API_DATA_FOLDER "output/
indoor_propagation_timevariant/Indoor_TimeVariant_All_TimeSteps");

  // Definition of dynamic sets for a moving Tx antenna. Alternative is to define the
 ID of group
  // where Tx antenna is mounted using GroupIDMounted of WinPropAntenna.
  WinPropDynAntenna =
 (WinProp_Dynamic_Antenna*)malloc(sizeof(WinProp_Dynamic_Antenna) * 3);
  if (WinPropDynAntenna != nullptr)
  {
   WinPropDynAntenna[0].Location = { 5.0f, 5.0f, 2.5f };
   WinPropDynAntenna[0].MovingDir = { 1.0f, 0.0f, 0.0f };
   WinPropDynAntenna[0].Azimuth = 0.f;
   WinPropDynAntenna[0].Downtilt = 0.f;
   WinPropDynAntenna[0].Roll = 0.f;
   WinPropDynAntenna[0].TimeStep = 0.f;
   WinPropDynAntenna[0].Velocity = 10.f;

   WinPropDynAntenna[1].Location = { 10.0f, 5.0f, 2.5f };
   WinPropDynAntenna[1].MovingDir = { 1.0f, 1.0f, 0.0f };
   WinPropDynAntenna[1].Azimuth = 0.f;
   WinPropDynAntenna[1].Downtilt = 0.f;
   WinPropDynAntenna[1].Roll = 0.f;
   WinPropDynAntenna[1].TimeStep = 0.5f;
   WinPropDynAntenna[1].Velocity = 10.f;

   WinPropDynAntenna[2].Location = { 10.0f, 10.0f, 2.5f };
   WinPropDynAntenna[2].MovingDir = { 0.0f, 1.0f, 0.0f };
   WinPropDynAntenna[2].Azimuth = 0.f;
   WinPropDynAntenna[2].Downtilt = 0.f;
   WinPropDynAntenna[2].Roll = 0.f;
   WinPropDynAntenna[2].TimeStep = 1.f;
   WinPropDynAntenna[2].Velocity = 10.f;

   WinPropAntenna.NrTimeVariantSets = 3;
   WinPropAntenna.TimeVariantSets = WinPropDynAntenna;
  }

  /* ----- Start prediction (including loop over all time instances) ------ */
  Error = WinProp_Predict(
   ProjectHandle,
   &WinPropAntenna,
   &WinPropArea,
   nullptr,
   0,
   nullptr,
   &WinPropMore,
   &PowerResult,
   &DelayResult,
   &LosResult,
   &RayMatrix,
```

```
    nullptr,
    nullptr,
    nullptr);

  if (Error == 0)
  {
   // write results to ASCII files
   char PowerOutputFile[200];
   sprintf(PowerOutputFile, API_DATA_FOLDER "output/indoor_propagation_timevariant/
%s_Power.txt", WinPropAntenna.Name);
   write_ascii(PowerResult, PowerOutputFile);

   char LoSOutputFile[200];
   sprintf(LoSOutputFile, API_DATA_FOLDER "output/indoor_propagation_timevariant/
%s_LOS.txt", WinPropAntenna.Name);
   write_ascii(LosResult, LoSOutputFile);

   char DelayOutputFile[200];
   sprintf(DelayOutputFile, API_DATA_FOLDER "output/indoor_propagation_timevariant/
%s_Delay.txt", WinPropAntenna.Name);
   write_ascii(DelayResult, DelayOutputFile);
  }

  WinProp_Close(ProjectHandle);

  // Free allocated memory
  if (WinPropDynAntenna != nullptr)
   free(WinPropDynAntenna);
 }


 return 0;
}

int _STD_CALL CallbackMessage(const char * Text)
{
 if (Text == nullptr)
  return 0;

 std::cout << "\n" << Text;

 return(0);
}

int _STD_CALL CallbackError(const char * Text, int Error)
{
 if (Text == nullptr)
  return 0;

 std::cout << "\n";

#ifdef __LINUX
 std::cout << "\033[31m" << "Error (" << Error << "): "; // highlight error in red
 color
#else
 HANDLE hConsole = GetStdHandle(STD_OUTPUT_HANDLE);
 SetConsoleTextAttribute(hConsole, FOREGROUND_RED);
 std::cout << "Error (" << Error << "): ";
#endif // __LINUX
 std::cout << Text;

#ifdef __LINUX
```

```
 std::cout << "\033[0m"; // highlight error in red color
#else
 SetConsoleTextAttribute(hConsole, FOREGROUND_RED | FOREGROUND_BLUE |
 FOREGROUND_GREEN);
#endif // __LINUX

 return 0;
}

int _STD_CALL CallbackProgress(int value, const char* text)
{
 char Line[200];

 sprintf(Line, "\n%d%% %s", value, text);
 std::cout << Line;

 return(0);
}

// Helper functions
void write_ascii(const WinProp_Result* Resultmatrix, const char* Filename)
{
 FILE* OutputFile = fopen(Filename,"w");
 if (OutputFile)
 {
  for (int t = 0; t < Resultmatrix->NrHeights; t++)
  {
   double Timestep = Resultmatrix->TimeSteps[t];
   for (int x = 0; x < Resultmatrix->Columns; x++)
   {
    for (int y = 0; y < Resultmatrix->Lines; y++)
    {
     /* Compute real coordinates. */
     double Coordinate_X = Resultmatrix->LowerLeftX + ((double)x + 0.5) *
 Resultmatrix->Resolution;
     double Coordinate_Y = Resultmatrix->LowerLeftY + ((double)y + 0.5) *
 Resultmatrix->Resolution;
     /* Check if pixel was computed or not */
     if (Resultmatrix->Matrix[t][x][y] > -1000)
      fprintf(OutputFile, "%.2f\t%.2f\t%.2f\t%.2f\n", Coordinate_X, Coordinate_Y,
 Timestep, Resultmatrix->Matrix[t][x][y]);
    }
   }
  }
  /* Close file. */
  fclose(OutputFile);
 }
 else
  printf("\nCould not open the File: %s for writing.\n",Filename);
}
```

2.6.11 Network Planning

Detailed Description

This is an example of how to use the WinProp API for network planning. The full example is distributed with the installation.

```
#include <stdio.h>
#include <string>
#include <iostream>
#include <cstring>

#include "network_planning.h"

#ifndef API_DATA_FOLDER
#define API_DATA_FOLDER "../../api/winprop/data/"
#endif // !API_DATA_FOLDER
#define SERVICES_LTE 13
#define MAX_TRX 3

int main(int argc, char** argv)
{
 /* ------------------ network planning parameters for LTE -------------------- */
 double AntennaX[MAX_TRX] = { 5.920, 63.860, 63.400 };
 double AntennaY[MAX_TRX] = { 6.240, -22.870, 12.610 };
 char AntennaName[MAX_TRX][500] = { "Site 1", "Site 2", "Site 3" };
 int MapIndex[MAX_TRX];

 WinProp_Result *MaxThroughput = NULL, *MaxSNIR = NULL;
 WinProp_Result PropResults[MAX_TRX];

 WinProp_Callback Callback;
 WinProp_Structure_Init_Callback(&Callback);

 WinProp_Antenna Antenna[MAX_TRX];
    for (size_t i = 0; i < MAX_TRX; i++)
    {
        WinProp_Structure_Init_Antenna(&Antenna[i]);
        WinProp_Structure_Init_Result(&PropResults[i]);
    }

 WinProp_Carrier Carrier;
 WinProp_Structure_Init_Carrier(&Carrier);

 int                 Signal_for_MIMO = -1, Stream_for_MIMO = -1;
 char                ProjectName[200];
 double              Frequency = 1800.0;
 int                 NrPredictionHeights = 1;
 double              PredictionHeightsMulti[2] = { 1.5, 5.2};

 int Error = 0;
 int ProjectHandle = 0;
 NrPredictionHeights
 = sizeof(PredictionHeightsMulti)/sizeof(PredictionHeightsMulti[0]);


 /* Open new network project based on .wst file */
 if (Error == 0)
 {
```

```
 Error = WinProp_Net_Project_Open_WST(&ProjectHandle,
API_DATA_FOLDER "airInterfaces/5G FDD.wst", NULL);
}

if (Error == 0)
{
 Error = WinProp_Net_Project_Para_Get(ProjectHandle, NET_PARA_PROJECTNAME, NULL,
NULL, ProjectName);
}

/* Set resolution for result matrix. */
if (Error == 0)
{
 double ParaValue = 1.0;
 Error = WinProp_Net_Project_Para_Set(ProjectHandle, NET_PARA_RESOLUTION,
&ParaValue, NULL, NULL);
}

/* Set size of area: Automatic mode. */
if (Error == 0)
{
 int IntValue = 0;
 Error = WinProp_Net_Project_Para_Set(ProjectHandle, NET_PARA_AREA_MODE, NULL,
&IntValue, NULL);
}

/* Set paths for additional output in WinProp file format. */
if (Error == 0)
{
 Error = WinProp_Net_Project_Para_Set(ProjectHandle, NET_PARA_OUTPUT_WINPROP, NULL,
NULL, API_DATA_FOLDER "output/network_planning");
}
/* Compute wave propagation for three transmitters                          */
/* Init. antennas and results */
int maxTRX = MAX_TRX;
int NrCarriers = 0;
Error = WinProp_Net_Project_Para_Get(ProjectHandle, NET_PARA_CARRIERS, NULL,
&NrCarriers, NULL);
if (Error == 0)
{
 if (NrCarriers < MAX_TRX)
  maxTRX = NrCarriers;
}

for (int Count = 0; Count < maxTRX; Count++)
{
 /* Determine Carrier ID. */
 int CarrID = 0;
 WinProp_Net_Carrier_Para_Get(ProjectHandle, Count, NET_PARA_CARRIER_INDEX, NULL,
&CarrID, NULL);

 /* Determine frequency for antenna. */
 WinProp_Net_Carrier_Para_Get(ProjectHandle, CarrID, NET_PARA_CARRIER_FREQ_DL,
&Frequency, NULL, NULL);

 /* Set properties now. */
 AntennaPropertiesSet(&Antenna[Count], AntennaX[Count], AntennaY[Count], 2.5,
Frequency, AntennaName[Count], Count + 1);

 // Init of carrier settings
 WinProp_Structure_Init_Carrier(&Carrier);
```

```
 // set carrier properties
 Carrier.CarrierID = CarrID;
 Carrier.SystemID = Signal_for_MIMO;
 Carrier.MimoID = Stream_for_MIMO;
 CarrierPropertiesSet(&Antenna[Count], &Carrier);
}
/* Compute wave propagation for all antennas. */
if (Error == 0)
{
 WavePropagation(maxTRX, Antenna, PropResults, NrPredictionHeights,
PredictionHeightsMulti,
  API_DATA_FOLDER "indoor/IndoorVectordatabase.idb");
}

// write propagation results to ascii files
if (Error == 0)
{
 for (int Count = 0; Count < maxTRX; Count++)
 {
  char PowerResultFile[600];
  sprintf(PowerResultFile, API_DATA_FOLDER "output/network_planning/%s
Power.txt",Antenna[Count].Name);
  write_ascii(&PropResults[Count], PowerResultFile);
 }
}
/* Now do network planning */
/* Add all propagation maps which have been computed before. */
if (Error == 0)
{
 for (int Count = 0; Count< maxTRX; Count++)
 {
  if (Error == 0)
  {
   Error = WinProp_Net_PropagationMap_Add(ProjectHandle, &MapIndex[Count],
&Antenna[Count], &PropResults[Count]);
  }
 }
}

if (Error == 0)
{
 char HeightString[500];
 sprintf(HeightString, "%s", "");

 /* Generate string with height values, e.g. a string like "1.5 2.5 3.5". */
 for (int Height = 0; Height<NrPredictionHeights; Height++)
 {
  /* Add current height to string. */
  char thisHeightStr[50];
  sprintf(thisHeightStr, "%.2f ", PredictionHeightsMulti[Height]);
  strcat(HeightString, thisHeightStr);
 }

 /* Send heights to WinProp API. */
 Error = WinProp_Net_Project_Para_Set(ProjectHandle, NET_PARA_HEIGHT_MULTIPLE, NULL,
NULL, HeightString);

 /* Start computation. */
 if (Error == 0)
 {
  Callback.Percentage = CallbackProgress;
  Callback.Message = CallbackMessage;
```

```
   Callback.Error = CallbackError;
   Error = WinProp_Net_Project_Compute(ProjectHandle, &Callback);
  }
 }

 // ------------------------------------------------------------------------
 // Retrieve results
 // ------------------------------------------------------------------------
 /* As an example: retrieve max. throughput (kbps) per pixel. */
 if (Error == 0)
 {
  Error = WinProp_Net_NetworkMap_Get(ProjectHandle, -1, NET_RESULT_MAX_THROUGHPUT,
 &MaxThroughput);
 }

 /* Write max. throughput result to ASCII file. */
 if (Error == 0)
 {
  char NameForOutput[200];
  sprintf(NameForOutput, API_DATA_FOLDER "output/network_planning/%s Max
 Throughput.txt", ProjectName);
  write_ascii(MaxThroughput, NameForOutput);
 }

 /* As another example: retrieve SNIR (dB) per pixel. */
 if (Error == 0)
 {
  Error = WinProp_Net_NetworkMap_Get(ProjectHandle, -1, NET_RESULT_SNIR, &MaxSNIR);
 }

 /* Write max. throughput result to ASCII file. */
 if (Error == 0)
 {
  char NameForOutput[200];
        sprintf(NameForOutput, API_DATA_FOLDER "output/network_planning/%s Max
 SNIR.txt", ProjectName);
        write_ascii(MaxSNIR, NameForOutput);
 }

 // ------------------------------------------------------------------------
 // Free Memory
 // ------------------------------------------------------------------------
 /* Free propagation results. */
 for (int Count = 0; Count < maxTRX; Count++)
  WinProp_FreeResult(&PropResults[Count]);
 /* Close network project. */
 Error = WinProp_Net_Project_Close(ProjectHandle);

 return 0;
}

int _STD_CALL CallbackMessage(const char * Text)
{
 if (Text == nullptr)
  return 0;

 std::cout << "\n" << Text;

 return(0);
}

int _STD_CALL CallbackError(const char * Text, int Error)
```

```
{
 if (Text == nullptr)
  return 0;

 std::cout << "\n";

#ifdef __LINUX
 std::cout << "\033[31m" << "Error (" << Error << "): "; // highlight error in red
 color
#else
 HANDLE hConsole = GetStdHandle(STD_OUTPUT_HANDLE);
 SetConsoleTextAttribute(hConsole, FOREGROUND_RED);
 std::cout << "Error (" << Error << "): ";
#endif // __LINUX
 std::cout << Text;

#ifdef __LINUX
 std::cout << "\033[0m"; // highlight error in red color
#else
 SetConsoleTextAttribute(hConsole, FOREGROUND_RED | FOREGROUND_BLUE |
 FOREGROUND_GREEN);
#endif // __LINUX

 return 0;
}

int _STD_CALL CallbackProgress(int value, const char* text)
{
 char Line[200];

 sprintf(Line, "\n%d%% %s", value, text);
 std::cout << Line;

 return(0);
}

// Helper functions
void write_ascii(const WinProp_Result* Resultmatrix, const char* Filename)
{
 FILE* OutputFile = fopen(Filename,"w");
 if (OutputFile)
 {
  /* Loop through WinPropall pixels. */
  for (int z = 0; z < Resultmatrix->NrHeights; z++)
  {
   double Coordinate_Z = Resultmatrix->Heights[z];
   for (int x = 0; x < Resultmatrix->Columns; x++)
   {
    for (int y = 0; y < Resultmatrix->Lines; y++)
    {
     /* Compute real coordinates. */
     double Coordinate_X = Resultmatrix->LowerLeftX + ((double)x + 0.5) *
 Resultmatrix->Resolution;
     double Coordinate_Y = Resultmatrix->LowerLeftY + ((double)y + 0.5) *
 Resultmatrix->Resolution;

     /* Check if pixel was computed or not */
     if (Resultmatrix->Matrix[z][x][y] > -1000)
      fprintf(OutputFile, "%.2f\t%.2f\t%.2f\t%.2f\n", Coordinate_X, Coordinate_Y,
 Coordinate_Z, Resultmatrix->Matrix[z][x][y]);
    }
   }
```

```
   }
   /* Close file. */
   fclose(OutputFile);
  }
  else
   printf("\nCould not open the File: %s for writing.\n",Filename);
}

void AntennaPropertiesSet(
 WinProp_Antenna *Antenna,
 double CoordinateX,
 double CoordinateY,
 double Height,
 double Frequency,
 const char *Name,
 int Id)
{
 Antenna->Longitude_X = CoordinateX;
 Antenna->Latitude_Y = CoordinateY;
 Antenna->Height = Height;
 Antenna->Power = 20.0;
 Antenna->PowerMode = WINPROP_POWER_OUTPUT;
 Antenna->Frequency = Frequency;
 sprintf(Antenna->Name, "%s", Name);
 Antenna->Model = WINPROP_MODEL_COST231;
 Antenna->Id = Id;
}

void CarrierPropertiesSet(WinProp_Antenna *Antenna, const WinProp_Carrier *Carrier)
{
 Antenna->Carriers.CarrierID = Carrier->CarrierID;
 Antenna->Carriers.SystemID = Carrier->SystemID;
 Antenna->Carriers.MimoID = Carrier->MimoID;
}

// Wave propagation
void WavePropagation(
 int NumberAntennas,
 const WinProp_Antenna *Antenna,
 WinProp_Result  *Result,
 int NrHeights,
 const double *Heights,
 const char* database)
{
 int                       PredictionHandle;
 WinProp_Area              Area;
 WinProp_Scenario          Scenario;
 WinProp_Callback          Callback;
 WinProp_Result            *DummyResult;
 int                       Error, Count;
 WinProp_Legend            Legend;
 WinProp_Additional     WinPropMore;
 WinProp_Propagation_Results OutputResults;
 // init
 Error = 0;
 PredictionHandle = 0;
 WinProp_Structure_Init_Area(&Area);
    WinProp_Structure_Init_Scenario(&Scenario);
    WinProp_Structure_Init_Callback(&Callback);
 WinProp_Structure_Init_Legend(&Legend);
 WinProp_Structure_Init_Additional(&WinPropMore);
 WinProp_Structure_Init_Propagation_Results(&OutputResults);
```

```
 // Open scenario
    Scenario.Scenario = WINPROP_SCENARIO_INDOOR;
    sprintf(Scenario.VectorDatabase, "%s", database);
 Callback.Percentage = CallbackProgress;
 Callback.Message = CallbackMessage;
 Callback.Error = CallbackError;

 /* Open new project. */
 if (Error == 0)
 {
  Error = WinProp_Open(&PredictionHandle, &Scenario, &Callback);
 }

 /* Define area */
 if (Error == 0)
 {
  /* Defintion of area. */
  Area.Heights = (double*)malloc(sizeof(double) * NrHeights);
  if (Area.Heights != nullptr)
        {
            Area.NrHeights = NrHeights;
   for (int C = 0; C < NrHeights; C++)
    Area.Heights[C] = Heights[C];
  }
  Area.LowerLeftX = -4.680001;
  Area.LowerLeftY = -44.200001;
  Area.UpperRightX = 101.399999;
  Area.UpperRightY = 42.799999;
  Area.Resolution = 1.0;
 }

 /* Write additional results */
 WinPropMore.OutputResults = &OutputResults;
 sprintf(OutputResults.ResultPath, "%s", API_DATA_FOLDER "output/
network_planning"); // Output data directory
 OutputResults.FieldStrength = 1;

 // ------------------------------------------------------------------------
 // Compute wave propagation
 // ------------------------------------------------------------------------
 for (Count = 0; Count < NumberAntennas; Count++)
 {
  if (Error == 0)
  {
   DummyResult = NULL;
   Error = WinProp_Predict(PredictionHandle, &Antenna[Count], &Area, NULL, 0, NULL,
 &WinPropMore, &DummyResult, NULL, NULL, NULL, NULL, NULL, NULL);
   if (Error == 0) {
    /* Copy result. */
    WinProp_CopyResult(&Result[Count], DummyResult);
   }
   else {
    /* Error during prediction. Print error message. */
    printf("\n\nSimulation returned with Error %d\n\n",Error);
   }
  }
 }
 // Free memory
 if (Area.Heights)
  free(Area.Heights);
 // Close project
 WinProp_Close(PredictionHandle);
```

```
}
```

2.6.12 Multi-threaded Network Planning

Detailed Description

This is an example of how to use the WinProp API for multi-threaded network planning with OpenMP. The full example is distributed with the installation.

```
#include <stdio.h>
#include <string>
#include <iostream>
#include <cstring>

#ifdef _OPENMP
#include <omp.h>
#endif

#include "network_planning_parallel.h"

#ifndef API_DATA_FOLDER
#define API_DATA_FOLDER "../../api/winprop/data/"
#endif // !API_DATA_FOLDER
#define SERVICES_LTE 13
#define MAX_TRX 3

#define MAX_THREADS 4


int main(int argc, char** argv)
{

 /* ----------------- network planning parameters for LTE ---------------------- */
 const char* TransmissionMode_Name[SERVICES_LTE] = {
  "QPSK - R=1_8",
  "QPSK - R=1_5",
  "QPSK - R=1_4",
  "QPSK - R=1_3",
  "QPSK - R=1_2",
  "QPSK - R=2_3",
  "QPSK - R=4_5",
  "16 QAM - R=1_2",
  "16 QAM - R=2_3",
  "16 QAM - R=4_5",
  "64 QAM - R=2_3",
  "64 QAM - R=3_4",
  "64 QAM - R=4_5" };
 double TransmissionMode_Bitrate[SERVICES_LTE] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0 };
 double TransmissionMode_SNIR[SERVICES_LTE] = { -5.4, -3.1, -2.2, -0.4, 1.6, 3.5,
 5.6, 7.0, 10.7, 11.8, 14.3, 16.1, 17.2 };
 int TransmissionMode_Coderate_K[SERVICES_LTE] = { 1, 1, 1, 1, 1, 2, 4, 1, 2, 4, 2,
 3, 4 };
```

```
int TransmissionMode_Coderate_N[SERVICES_LTE] = { 8, 5, 4, 3, 2, 3, 5, 2, 3, 5, 3,
4, 5 };
int TransmissionMode_MCS[SERVICES_LTE] = { 2, 2, 2, 2, 2, 2, 2, 4, 4, 4, 6, 6, 6 };
double TransmissionMode_Backoff[SERVICES_LTE] = { 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
3.0, 3.0, 3.0, 3.0, 3.0, 3.0 };

double AntennaX[MAX_TRX] = { 5.920, 63.860, 63.400 };
double AntennaY[MAX_TRX] = { 6.240, -22.870, 12.610 };
int AntennaCarrier[MAX_TRX] = { 1, 2, 2 };
char AntennaName[MAX_TRX][500] = { "Site 1", "Site 2", "Site 3" };
int MapIndex[MAX_TRX];

WinProp_Result *MaxThroughput = NULL;
WinProp_Result *MaxSNIR = NULL;

   WinProp_Result PropResults[MAX_TRX];
   WinProp_Antenna Antenna[MAX_TRX];

   for (int Count = 0; Count < MAX_TRX; Count++)
   {
       /* Init of antenna */
       WinProp_Structure_Init_Antenna(&Antenna[Count]);

       /* Init of result */
       WinProp_Structure_Init_Result(&PropResults[Count]);
   }

WinProp_Callback Callback;
WinProp_Structure_Init_Callback(&Callback);

WinProp_Carrier Carrier;
WinProp_Structure_Init_Carrier(&Carrier);

int                 Signal_for_MIMO = -1, Stream_for_MIMO = -1;
char                ProjectName[200];
double              Frequency = 1800.0;
int                 NrPredictionHeights = 1;
double              PredictionHeightsMulti[2] = { 1.5, 5.2};

int Error = 0;
int ProjectHandle = 0;
NrPredictionHeights
= sizeof(PredictionHeightsMulti)/sizeof(PredictionHeightsMulti[0]);

/* ------------ Create new network project: LTE air interface ---------------- */
if (Error == 0)
{
 Error = WinProp_Net_Project_Open(&ProjectHandle, NET_AIRINTERFACE_LTE_GENERIC,
NULL);
}

// Enable computation of network planning using multiple threads
if (Error == 0)
{
 int IntValue = MAX_THREADS;
 Error = WinProp_Net_Project_Para_Set(ProjectHandle, NET_SETTINGS_NUMBER_THREADS,
NULL, &IntValue, NULL);
}

// TDD mode
if (Error == 0)
{
```

```
 int IntValue = 1;
 Error = WinProp_Net_Project_Para_Set(ProjectHandle, NET_PARA_DUPLEX_SUBMODE, NULL,
&IntValue, NULL);
}

/* Set name of project (optional). */
if (Error == 0) {
 sprintf(ProjectName, "%s", "LTE Project");
 Error = WinProp_Net_Project_Para_Set(ProjectHandle, NET_PARA_PROJECTNAME, NULL,
NULL, ProjectName);
}

/* Retreive carrier separation (must be 20 MHz). */
double CarrierSeparation = 0.;
if (Error == 0)
{
 Error = WinProp_Net_Project_Para_Get(ProjectHandle, NET_PARA_CARRIER_SEPARATION,
&CarrierSeparation, NULL, NULL);
}

/* Add some carriers. */
if (Error == 0)
{
 CarrierSeparation *= 0.000001;
 for (int CurrentCarrier = 0; CurrentCarrier < 3; CurrentCarrier++)
 {
  double FrequencyUL = 1930.0 + (CurrentCarrier * CarrierSeparation);
  double FrequencyDL = 2120.0 + (CurrentCarrier * CarrierSeparation);
  Error = WinProp_Net_Carrier_Add(ProjectHandle, CurrentCarrier + 1);
  Error = WinProp_Net_Carrier_Para_Set(
   ProjectHandle, CurrentCarrier + 1, NET_PARA_CARRIER_FREQ_DL, &FrequencyUL, NULL,
NULL);
  Error = WinProp_Net_Carrier_Para_Set(
   ProjectHandle, CurrentCarrier + 1, NET_PARA_CARRIER_FREQ_UL, &FrequencyDL, NULL,
NULL);
 }
}

/* Check if number of carriers is correct. */
if (Error == 0)
{
 int NrCarriers = 0;
 Error = WinProp_Net_Project_Para_Get(ProjectHandle, NET_PARA_CARRIERS, NULL,
&NrCarriers, NULL);
 if (Error == 0)
 {
  if (NrCarriers != 3)
   Error = 1;
 }
}

/* Add now 13 transmission modes for LTE. */
if (Error == 0)
{
 for (int CurrentService = 0; CurrentService < SERVICES_LTE; CurrentService++)
 {
  /* Add new service. */
  char ServiceName[500];
  sprintf(ServiceName, "%s", TransmissionMode_Name[CurrentService]);
  Error = WinProp_Net_TransmissionMode_Add(ProjectHandle, ServiceName,
CurrentService);
```

```
  /* Set position/priority. */
  int IntValue = SERVICES_LTE - CurrentService;
  Error = WinProp_Net_TransmissionMode_Para_Set(
   ProjectHandle, CurrentService, NET_PARA_TRANS_MODE_POSITION, NULL, &IntValue,
NULL);

  /* Set bitrate. */
  double ParaValue = TransmissionMode_Bitrate[CurrentService];
  Error = WinProp_Net_TransmissionMode_Para_Set(
   ProjectHandle, CurrentService, NET_PARA_TRANS_MODE_BITRATE_DL, &ParaValue, NULL,
NULL);
  Error = WinProp_Net_TransmissionMode_Para_Set(
   ProjectHandle, CurrentService, NET_PARA_TRANS_MODE_BITRATE_UL, &ParaValue, NULL,
NULL);

  /* Set code rate. */
  IntValue = TransmissionMode_Coderate_K[CurrentService];
  Error = WinProp_Net_TransmissionMode_Para_Set(
   ProjectHandle, CurrentService, NET_PARA_TRANS_MODE_CODERATE_K_DL, NULL,
&IntValue, NULL);
  Error = WinProp_Net_TransmissionMode_Para_Set(
   ProjectHandle, CurrentService, NET_PARA_TRANS_MODE_CODERATE_K_UL, NULL,
&IntValue, NULL);
  IntValue = TransmissionMode_Coderate_N[CurrentService];
  Error = WinProp_Net_TransmissionMode_Para_Set(
   ProjectHandle, CurrentService, NET_PARA_TRANS_MODE_CODERATE_N_DL, NULL,
&IntValue, NULL);
  Error = WinProp_Net_TransmissionMode_Para_Set(
   ProjectHandle, CurrentService, NET_PARA_TRANS_MODE_CODERATE_N_UL, NULL,
&IntValue, NULL);

  /* Set number of resource blocks. */
  IntValue = 1;
  Error = WinProp_Net_TransmissionMode_Para_Set(
   ProjectHandle, CurrentService, NET_PARA_TRANS_MODE_RESOURCE_BLOCKS_DL, NULL,
&IntValue, NULL);
  Error = WinProp_Net_TransmissionMode_Para_Set(
   ProjectHandle, CurrentService, NET_PARA_TRANS_MODE_RESOURCE_BLOCKS_UL, NULL,
&IntValue, NULL);

  /* Set overhead ratio. */
  ParaValue = 11.1;
  Error = WinProp_Net_TransmissionMode_Para_Set(
   ProjectHandle, CurrentService, NET_PARA_TRANS_MODE_OVERHEAD_RATIO_UL, &ParaValue,
NULL, NULL);
  Error = WinProp_Net_TransmissionMode_Para_Set(
   ProjectHandle, CurrentService, NET_PARA_TRANS_MODE_OVERHEAD_RATIO_DL, &ParaValue,
NULL, NULL);

  /* Set required SNIR. */
  ParaValue = TransmissionMode_SNIR[CurrentService];
  Error = WinProp_Net_TransmissionMode_Para_Set(
   ProjectHandle, CurrentService, NET_PARA_TRANS_MODE_SNIR_DL, &ParaValue, NULL,
NULL);
  Error = WinProp_Net_TransmissionMode_Para_Set(
   ProjectHandle, CurrentService, NET_PARA_TRANS_MODE_SNIR_UL, &ParaValue, NULL,
NULL);

  /* Set modulation scheme. */
  IntValue = TransmissionMode_MCS[CurrentService];
  Error = WinProp_Net_TransmissionMode_Para_Set(
   ProjectHandle, CurrentService, NET_PARA_TRANS_MODE_MODULATION_UL, NULL,
&IntValue, NULL);
```

```
  Error = WinProp_Net_TransmissionMode_Para_Set(
   ProjectHandle, CurrentService, NET_PARA_TRANS_MODE_MODULATION_DL, NULL,
&IntValue, NULL);

  /* Set power backoff. */
  ParaValue = TransmissionMode_Backoff[CurrentService];
  Error = WinProp_Net_TransmissionMode_Para_Set(
   ProjectHandle, CurrentService, NET_PARA_TRANS_MODE_POWER_BACKOFF_UL, &ParaValue,
NULL, NULL);
  Error = WinProp_Net_TransmissionMode_Para_Set(
   ProjectHandle, CurrentService, NET_PARA_TRANS_MODE_POWER_BACKOFF_DL, &ParaValue,
NULL, NULL);

  /* Get bitrate in kBit/s for current transmission mode. */
  /* Bitrate is automatically computed based on parameters previously set and
general air interface parameters. */
  Error = WinProp_Net_TransmissionMode_Para_Get(
   ProjectHandle, CurrentService, NET_PARA_TRANS_MODE_BITRATE_DL, &ParaValue, NULL,
NULL);
  TransmissionMode_Bitrate[CurrentService] = ParaValue;
 }
}

/* Check if number of services is correct. */
if (Error == 0)
{
 int NrServices = 0;
 Error = WinProp_Net_Project_Para_Get(ProjectHandle, NET_PARA_SERVICES, NULL,
&NrServices, NULL);
 if (Error == 0)
 {
  if (NrServices != SERVICES_LTE)
   Error = 1;
 }
}

/* Set minimum required SNIR. */
if (Error == 0)
{
 double ParaValue = -5.4;
 Error = WinProp_Net_Project_Para_Set(ProjectHandle,
NET_PARA_CELL_ASSIGNMENT_MIN_REQ_SNIR, &ParaValue, NULL, NULL);
}

/* Disable min power criterion. */
if (Error == 0)
{
 int IntValue = 0;
 Error = WinProp_Net_Project_Para_Set(ProjectHandle,
NET_PARA_CELL_ASSIGNMENT_MIN_REQ_POWER_USED, NULL, &IntValue, NULL);
}

/* Set cell assignment mode. */
if (Error == 0)
{
 int IntValue = 0;
 Error = WinProp_Net_Project_Para_Set(ProjectHandle, NET_PARA_CELL_ASSIGNMENT_MODE,
NULL, &IntValue, NULL);
}

/* Set cell assignment signals. */
if (Error == 0)
```

```
{
 int IntValue = 0;
 Error = WinProp_Net_Project_Para_Set(ProjectHandle,
NET_PARA_CELL_ASSIGNMENT_SIGNALS, NULL, &IntValue, NULL);
}

/* Set resolution for result matrix. */
if (Error == 0)
{
 double ParaValue = 1.0;
 Error = WinProp_Net_Project_Para_Set(ProjectHandle, NET_PARA_RESOLUTION,
&ParaValue, NULL, NULL);
}

/* Set size of area: Automatic mode. */
if (Error == 0)
{
 int IntValue = 0;
 Error = WinProp_Net_Project_Para_Set(ProjectHandle, NET_PARA_AREA_MODE, NULL,
&IntValue, NULL);
}

/* Set paths for additional output in WinProp file format. */
if (Error == 0)
{
 Error = WinProp_Net_Project_Para_Set(ProjectHandle, NET_PARA_OUTPUT_WINPROP, NULL,
NULL, API_DATA_FOLDER "output/network_planning_parallel/network");
}

/* Compute wave propagation for three transmitters                          */
/* Init. antennas and results */
for (int Count = 0; Count < MAX_TRX && Error == 0; Count++)
{
 /* Determine frequency for antenna. */
 Frequency = 2120.0 + ((AntennaCarrier[Count] - 1) * CarrierSeparation);

 /* Set properties now. */
 AntennaPropertiesSet(&Antenna[Count], AntennaX[Count], AntennaY[Count], 2.5,
Frequency, AntennaName[Count], Count + 1);

 // Init of carrier settings
 WinProp_Structure_Init_Carrier(&Carrier);

 // set carrier properties
 Carrier.CarrierID = AntennaCarrier[Count];
 Carrier.SystemID = Signal_for_MIMO;
 Carrier.MimoID = Stream_for_MIMO;
 CarrierPropertiesSet(&Antenna[Count], &Carrier);
}
/* Compute wave propagation for all antennas. */
if (Error == 0)
{
 WavePropagation(
  MAX_TRX,
  Antenna,
  PropResults,
  NrPredictionHeights,
  PredictionHeightsMulti,
  API_DATA_FOLDER "indoor/IndoorVectordatabase.idb");
}

/* Now do network planning */
```

```
/* Add all propagation maps which have been computed before. */
if (Error == 0)
{
 for (int Count = 0; Count<MAX_TRX; Count++)
 {
  if (Error == 0)
  {
   Error = WinProp_Net_PropagationMap_Add(ProjectHandle, &MapIndex[Count],
&Antenna[Count], &PropResults[Count]);
  }
 }
}

if (Error == 0)
{
 char HeightString[500];
 sprintf(HeightString, "%s", "");

 /* Generate string with height values, e.g. a string like "1.5 2.5 3.5". */
 for (int Height = 0; Height<NrPredictionHeights; Height++)
 {
  /* Add current height to string. */
  char thisHeightStr[50];
  sprintf(thisHeightStr, "%.2f ", PredictionHeightsMulti[Height]);
  strcat(HeightString, thisHeightStr);
 }

 /* Send heights to WinProp API. */
 Error = WinProp_Net_Project_Para_Set(ProjectHandle, NET_PARA_HEIGHT_MULTIPLE, NULL,
NULL, HeightString);

 /* Start computation. */
 if (Error == 0)
 {
  Callback.Percentage = CallbackProgress;
  Callback.Message = CallbackMessage;
  Callback.Error = CallbackError;
  Error = WinProp_Net_Project_Compute(ProjectHandle, &Callback);
 }
}

// ------------------------------------------------------------------------
// Retrieve results
// ------------------------------------------------------------------------
/* As an example: retrieve max. throughput (kbps) per pixel. */
if (Error == 0)
{
 Error = WinProp_Net_NetworkMap_Get(ProjectHandle, -1, NET_RESULT_MAX_THROUGHPUT,
&MaxThroughput);
}

/* Write max. throughput result to ASCII file. */
if (Error == 0)
{
 char NameForOutput[200];
 sprintf(NameForOutput, API_DATA_FOLDER "output/network_planning_parallel/network/%s
Max Throughput.txt", ProjectName);
 write_ascii(MaxThroughput, NameForOutput);
}

/* As another example: retrieve SNIR (dB) per pixel. */
if (Error == 0)
```

```
 {
  Error = WinProp_Net_NetworkMap_Get(ProjectHandle, -1, NET_RESULT_SNIR, &MaxSNIR);
 }

 /* Write max. throughput result to ASCII file. */
 if (Error == 0)
 {
  char NameForOutput[200];
       sprintf(NameForOutput, API_DATA_FOLDER "output/network_planning_parallel/
network/%s Max SNIR.txt", ProjectName);
       write_ascii(MaxSNIR, NameForOutput);
 }

 // ----------------------------------------------------------------------------
 // Free Memory
 // ----------------------------------------------------------------------------
 /* Free propagation results. */
 for (int Count = 0; Count<MAX_TRX; Count++)
  WinProp_FreeResult(&PropResults[Count]);
 /* Close network project. */
 Error = WinProp_Net_Project_Close(ProjectHandle);

 return 0;
}

int _STD_CALL CallbackMessage(const char * Text)
{
 if (Text == nullptr)
  return 0;

 std::cout << "\n" << Text;

 return(0);
}

int _STD_CALL CallbackError(const char * Text, int Error)
{
 if (Text == nullptr)
  return 0;

 std::cout << "\n";

#ifdef __LINUX
 std::cout << "\033[31m" << "Error (" << Error << "): "; // highlight error in red
 color
#else
 HANDLE hConsole = GetStdHandle(STD_OUTPUT_HANDLE);
 SetConsoleTextAttribute(hConsole, FOREGROUND_RED);
 std::cout << "Error (" << Error << "): ";
#endif // __LINUX
 std::cout << Text;

#ifdef __LINUX
 std::cout << "\033[0m"; // highlight error in red color
#else
 SetConsoleTextAttribute(hConsole, FOREGROUND_RED | FOREGROUND_BLUE |
 FOREGROUND_GREEN);
#endif // __LINUX

 return 0;
}
```

```
int _STD_CALL CallbackProgress(int value, const char* text)
{
 char Line[200];

 sprintf(Line, "\n%d%% %s", value, text);
 std::cout << Line;

 return(0);
}

// Helper functions
void write_ascii(const WinProp_Result* Resultmatrix, const char* Filename) {
 FILE* OutputFile = fopen(Filename,"w");
 if (OutputFile)
 {
  /* Loop through WinPropall pixels. */
  for (int z = 0; z < Resultmatrix->NrHeights; z++)
  {
   double Coordinate_Z = Resultmatrix->Heights[z];
   for (int x = 0; x < Resultmatrix->Columns; x++)
   {
    for (int y = 0; y < Resultmatrix->Lines; y++)
    {
     /* Compute real coordinates. */
     double Coordinate_X = Resultmatrix->LowerLeftX + ((double)x + 0.5) *
 Resultmatrix->Resolution;
     double Coordinate_Y = Resultmatrix->LowerLeftY + ((double)y + 0.5) *
 Resultmatrix->Resolution;

     /* Check if pixel was computed or not */
     if (Resultmatrix->Matrix[z][x][y] > -1000)
      fprintf(OutputFile, "%.2f\t%.2f\t%.2f\t%.2f\n", Coordinate_X, Coordinate_Y,
 Coordinate_Z, Resultmatrix->Matrix[z][x][y]);
    }
   }
  }
  /* Close file. */
  fclose(OutputFile);
 }
 else
  printf("\nCould not open the File: %s for writing.\n",Filename);
}

void AntennaPropertiesSet(
 WinProp_Antenna *Antenna,
 double CoordinateX,
 double CoordinateY,
 double Height,
 double Frequency,
 const char *Name,
 int Id)
{
 Antenna->Longitude_X = CoordinateX;
 Antenna->Latitude_Y = CoordinateY;
 Antenna->Height = Height;
 Antenna->Power = 20.0;
 Antenna->PowerMode = WINPROP_POWER_OUTPUT;
 Antenna->Frequency = Frequency;
 sprintf(Antenna->Name, "%s", Name);
 Antenna->Model = WINPROP_MODEL_COST231;
 Antenna->Id = Id;
}
```

```
void CarrierPropertiesSet(WinProp_Antenna *Antenna, const WinProp_Carrier *Carrier)
{
 Antenna->Carriers.CarrierID = Carrier->CarrierID;
 Antenna->Carriers.SystemID = Carrier->SystemID;
 Antenna->Carriers.MimoID = Carrier->MimoID;
}

// Wave propagation
void WavePropagation(
 int NumberAntennas,
 const WinProp_Antenna *Antenna,
 WinProp_Result *Result,
 int NrHeights,
 const double *Heights,
 const char* database)
{
 // init
 int Error = 0;
 WinProp_Area Area;
 WinProp_Structure_Init_Area(&Area);

 // Open scenario
 WinProp_Scenario Scenario;
 WinProp_Structure_Init_Scenario(&Scenario);
 Scenario.Scenario = WINPROP_SCENARIO_INDOOR;
 sprintf(Scenario.VectorDatabase, "%s", database);

 // set-up additional result writing
 WinProp_Propagation_Results OutputResults;
 WinProp_Structure_Init_Propagation_Results(&OutputResults);
 sprintf(OutputResults.ResultPath, "%s", API_DATA_FOLDER "output/
network_planning_parallel/propagation"); // Output data directory
 OutputResults.FieldStrength = 1;
 OutputResults.PathLoss = 1;

 WinProp_Additional  WinPropMore;
 WinProp_Structure_Init_Additional(&WinPropMore);
 WinPropMore.OutputResults = &OutputResults;

 WinProp_Callback Callback;
 WinProp_Structure_Init_Callback(&Callback);

 Callback.Percentage = CallbackProgress;
 Callback.Message = CallbackMessage;
 Callback.Error = CallbackError;

 /* Define area */
 if (Error == 0)
 {
  /* Definition of area. */
  Area.Heights = (double*)malloc(sizeof(double) * NrHeights);
  if (Area.Heights != nullptr)
        {
            Area.NrHeights = NrHeights;
   for (int C = 0; C < NrHeights; C++)
    Area.Heights[C] = Heights[C];
  }
  Area.LowerLeftX = -4.680001;
  Area.LowerLeftY = -44.200001;
  Area.UpperRightX = 101.399999;
  Area.UpperRightY = 42.799999;
```

```
  Area.Resolution = 1.0;
 }

 // ------------------------------------------------------------------------
 // Compute wave propagation
 // ------------------------------------------------------------------------
 // Set the number of parallel threads.
#ifdef _OPENMP
 omp_set_num_threads(MAX_THREADS);
#endif

 // As an example we're computing the same antenna location in the same environment
 with 16 different frequencies.
#pragma omp parallel for schedule(dynamic,1) shared(Area, Antenna, Scenario,
 Callback, Result)
 for (int Count = 0; Count < NumberAntennas; Count++)
 {
  /* Open new project. */
  int PredictionHandle = 0;
  if (Error == 0)
  {
   Error = WinProp_Open(&PredictionHandle, &Scenario, &Callback);
  }

  if (Error == 0)
  {
   WinProp_Result* DummyResult = nullptr;
   Error = WinProp_Predict(PredictionHandle, &Antenna[Count], &Area, NULL, 0, NULL,
 &WinPropMore, &DummyResult, NULL, NULL, NULL, NULL, NULL, NULL);
   if (Error == 0) {
    /* Copy result. */
    WinProp_CopyResult(&Result[Count], DummyResult);
   }
   else {
    /* Error during prediction. Print error message. */
    printf("\n\nSimulation returned with Error %d\n\n",Error);
   }
  }

  // Close project
  WinProp_Close(PredictionHandle);
 }

 // Write ASCII results
 for (int Count = 0; Count < NumberAntennas; Count++)
 {
  char PropOutputFile[500];
  sprintf(PropOutputFile, API_DATA_FOLDER "output/network_planning_parallel/
propagation/%s Power.txt",Antenna[Count].Name);
  write_ascii(&Result[Count], PropOutputFile);
 }

 // Free memory
 if (Area.Heights)
  free(Area.Heights);
}
```

2.6.13 Monte Carlo analysis for 5G

Detailed Description

This is an example of how to use 5G Monte-Carlo analysis capability with WinProp API. The full example is distributed with the installation.

```
#include <stdio.h>
#include <string>
#include <iostream>
#include <cstring>

#include "monte_carlo_5g.h"

#ifndef API_DATA_FOLDER
#define API_DATA_FOLDER "../../api/winprop/data/"
#endif // !API_DATA_FOLDER
#define SERVICES_5G 7
#define MAX_CELLS 3

void AntennaPropertiesSet(
    WinProp_Antenna* Antenna,
    double CoordinateX,
    double CoordinateY,
    double Height,
    double Frequency,
    const char* Name,
    double Power,
    WINPROP_POWER_MODE PowerMode,
    WINPROP_MODEL_INDOOR Model,
    double Azimuth,
    double Downtilt,
    int Id)
{
    Antenna->Longitude_X = CoordinateX;
    Antenna->Latitude_Y = CoordinateY;
    Antenna->Height = Height;
    Antenna->Power = Power;
    Antenna->PowerMode = PowerMode;
    Antenna->Frequency = Frequency;
    sprintf(Antenna->Name, "%s", Name);
    Antenna->Model = Model;
    Antenna->Azimuth = Azimuth;
    Antenna->Downtilt = Downtilt;
    Antenna->Id = Id;

}

int main(int argc, char** argv)
{

 // ------ General and Propagation and Network Planing Parameters ------ //
 WinProp_ParaMain    GeneralParameters;
 WinProp_Pattern     WinPropPattern;
 WinProp_Antenna     Antenna[MAX_CELLS];
 WinProp_Carrier     Carrier[MAX_CELLS];
 WinProp_Propagation_Results OutputResults;

 // Propagation results
```

```
 WinProp_Result      PropResults[MAX_CELLS];

 int                 ProjectHandle = 0, Error = 0;
 int                 ParaValueInt = 0;       // to set int parameters
 double              ParaValueDouble = 0.;   // to set double  parameters

 // Prediction heights
 int                 NrPredictionHeights = 1;
 double              PredictionHeights[1] = {1.5};

 // ---------------------- Initialisation of parameters ---------------- //
 WinProp_Structure_Init_ParameterMain(&GeneralParameters);
 WinProp_Structure_Init_Pattern(&WinPropPattern);
 WinProp_Structure_Init_Propagation_Results(&OutputResults);
 for (int cell = 0; cell < MAX_CELLS; cell++)
 {
  WinProp_Structure_Init_Antenna(&Antenna[cell]);
  WinProp_Structure_Init_Carrier(&Carrier[cell]);
  WinProp_Structure_Init_Result(&PropResults[cell]);
 }

 // ---------------- Definition of parameters -------------------------- //
 // Definition of scenario.
 const char* my_odb_database = API_DATA_FOLDER "outdoor/Frankfurt"; // name of .odb
 database without extensions
 GeneralParameters.ScenarioMode = SCENARIOMODE_URBAN; // Urban prediction
 GeneralParameters.PredictionModelUrban = PREDMODEL_UDP; // Use Dominant Path Model

 // Definition of prediction area.
 GeneralParameters.UrbanLowerLeftX = 475080.0;
 GeneralParameters.UrbanLowerLeftY = 551860.0;
 GeneralParameters.UrbanUpperRightX = 476840.0;
 GeneralParameters.UrbanUpperRightY = 553200.0;

 // Copy coordinates to prediction area of second model (not yet used).
 GeneralParameters.RuralLowerLeftX = GeneralParameters.UrbanLowerLeftX;
 GeneralParameters.RuralLowerLeftY = GeneralParameters.UrbanLowerLeftY;
 GeneralParameters.RuralUpperRightX = GeneralParameters.UrbanUpperRightX;
 GeneralParameters.RuralUpperRightY = GeneralParameters.UrbanUpperRightY;

 // Size of matrix with results.
 GeneralParameters.Resolution = 10.0;                        // Resolution in meter
 GeneralParameters.NrLayers = NrPredictionHeights;           // Number of prediction
 heights
 GeneralParameters.PredictionHeights = PredictionHeights;    // Prediction height in
 meter


 // Building vector data and topography.
 GeneralParameters.BuildingsMode = BUILDINGSMODE_BINARY; // load a .odb database
 sprintf(GeneralParameters.BuildingsName, "%s", my_odb_database);

 // Antenna patten (will be used after the propagation analysis to mask the omni-
propagation
 // predictions)
 WinPropPattern.Mode = PATTERN_MODE_FILE;   // Load pattern from file
 sprintf(WinPropPattern.Filename, "%s", API_DATA_FOLDER "antennas/
Beams2Control4Data.ffe"); // Pattern file

 // Write additional results
 GeneralParameters.OutputResults = &OutputResults;
 sprintf(OutputResults.ResultPath, "%s", API_DATA_FOLDER "output/monte_carlo_5g");
```

```
OutputResults.FieldStrength = 1;
OutputResults.PathLoss = 1;

// -------------------- Network Planning Project ----------------------- //
// Create a new network planning project based on 5G FDD air interface
if (Error == 0)
{
 Error = WinProp_Net_Project_Open(&ProjectHandle, NET_AIRINTERFACE_5G_FDD_GENERIC,
NULL);
}

// Set a name for the project.
if (Error == 0)
{
 char ProjectName[200];
 sprintf(ProjectName, "%s", "MonteCarlo-5G");
 Error = WinProp_Net_Project_Para_Set(ProjectHandle, NET_PARA_PROJECTNAME, NULL,
NULL, ProjectName);
}

// --------------------------  Carriers  ------------------------------- //
// Add 3 carriers in n1 band.
if (Error == 0)
{
 double FrequencyUL = 0.;
 double FrequencyDL = 0.;
 double CarrierSeparation = 20.;
 for (int CurrentCarrier = 0; CurrentCarrier < 3; CurrentCarrier++)
 {
  FrequencyUL = 1930.0 + (CurrentCarrier * CarrierSeparation);
  FrequencyDL = 2120.0 + (CurrentCarrier * CarrierSeparation);
  Error = WinProp_Net_Carrier_Add(ProjectHandle, CurrentCarrier + 1);

  Error = WinProp_Net_Carrier_Para_Set(
   ProjectHandle, CurrentCarrier + 1, NET_PARA_CARRIER_FREQ_DL, &FrequencyDL, NULL,
NULL);
  Error = WinProp_Net_Carrier_Para_Set(
   ProjectHandle, CurrentCarrier + 1, NET_PARA_CARRIER_FREQ_UL, &FrequencyUL, NULL,
NULL);
 }
}

// Check if number of carriers is correct.
if (Error == 0)
{
 int NrCarriers = 0;
 Error = WinProp_Net_Project_Para_Get(ProjectHandle, NET_PARA_CARRIERS, NULL,
&NrCarriers, NULL);
 if (Error == 0)
 {
  if (NrCarriers != 3)
   Error = 1;
 }
}

// --------------------------  antennas  ------------------------------- //
// Position and configuration of the antennas (cells).
char AntennaName[MAX_CELLS][500] = { "Ant 3", "Ant 2", "Ant 1" };
double SiteX = 475500.;
double SiteY = 552897.;
double SiteZ = 12.;
double Power = 40.;  // Power in dBm
```

```
double Azmith[MAX_CELLS] = { 240., 120., 0. };

// same frequencies in MHz as defined for the carriers
double Frequency[MAX_CELLS] = { 2160., 2140., 2120. };

double Downtilt = 0.;
int AntennaCarrier[MAX_CELLS] = { 3, 2, 1 };

   if (Error == 0)
   {
 for (int cell = 0; cell < MAX_CELLS; cell++)
 {
  // Set antenna properties now.
  AntennaPropertiesSet(
   &Antenna[cell], SiteX, SiteY, SiteZ, Frequency[cell], AntennaName[cell],
   Power, WINPROP_POWER_OUTPUT, WINPROP_MODEL_DPM, Azmith[cell], Downtilt, cell +
1);

  Carrier[cell].CarrierID = AntennaCarrier[cell];
  // Define two data beams for each cell (network planning analysis)
  Carrier[cell].NrAntennaBeams = 2;
  CarrierPropertiesSet(&Antenna[cell], &Carrier[cell]);
 }
}

// -------------------------  Transmission modes  -------------------------- //
// Define some (only 7) of the 5G transmission modes
if (Error == 0)
{
 const char*         TransmissionMode_Name[SERVICES_5G] = {
                                  "00 - QPSK - R 120",
           "04 - QPSK - R 602",
           "08 - 16 QAM - R 553",
           "12 - 64 QAM - R 517",
           "18 - 64 QAM - R 822",
           "21 - 256 QAM - R 711",
           "27 - 256 QAM - R 948" };
 double              TransmissionMode_Bitrate[SERVICES_5G] = { 0, 0, 0, 0, 0, 0,
0 };
 double              TransmissionMode_SNIR[SERVICES_5G] = { -3.0, 2, 7.9, 13.0,
17.5, 22.0, 27.0 };
 int                 TransmissionMode_Coderate_K[SERVICES_5G] = { 120, 602, 553,
517, 822, 711, 948 };
 int                 TransmissionMode_Coderate_N[SERVICES_5G] = { 1024, 1024, 1024,
1024, 1024, 1024, 1024 };
 int                 TransmissionMode_MCS[SERVICES_5G] = { 2, 2, 4, 6, 6, 8, 8 };
 double              TransmissionMode_Backoff[SERVICES_5G] = { 0.0, 0.0, 3.0, 3.0,
3.0, 3.0, 3.0 };
 int                 TransmissionMode_Resource_Blocks_DL[SERVICES_5G] = { 1, 1, 1,
1, 1, 1, 1 };
 int                 TransmissionMode_Resource_Blocks_UL[SERVICES_5G] = { 1, 1, 1,
1, 1, 1, 1 };
 double              TransmissionMode_Overhead_Ratio_DL[SERVICES_5G] = { 14., 14.,
14., 14., 14., 14., 14. };
 double              TransmissionMode_Overhead_Ratio_UL[SERVICES_5G] = { 8., 8., 8.,
8., 8., 8., 8. };

 for (int CurrentService = 0; CurrentService < SERVICES_5G; CurrentService++)
 {

           // Add new service.
           char ServiceName[100];
```

```
  sprintf(ServiceName, "%s", TransmissionMode_Name[CurrentService]);
  Error = WinProp_Net_TransmissionMode_Add(ProjectHandle, ServiceName,
CurrentService);

  // Set position/priority.
  ParaValueInt = SERVICES_5G - CurrentService;
  Error = WinProp_Net_TransmissionMode_Para_Set(
   ProjectHandle, CurrentService, NET_PARA_TRANS_MODE_POSITION, NULL, &ParaValueInt,
NULL);

  // Set bitrate.
  ParaValueDouble = TransmissionMode_Bitrate[CurrentService];
  Error = WinProp_Net_TransmissionMode_Para_Set(
   ProjectHandle, CurrentService, NET_PARA_TRANS_MODE_BITRATE_DL, &ParaValueDouble,
NULL, NULL);
  Error = WinProp_Net_TransmissionMode_Para_Set(
   ProjectHandle, CurrentService, NET_PARA_TRANS_MODE_BITRATE_UL, &ParaValueDouble,
NULL, NULL);

  // Set code rate.
  ParaValueInt = TransmissionMode_Coderate_K[CurrentService];
  Error = WinProp_Net_TransmissionMode_Para_Set(
   ProjectHandle, CurrentService, NET_PARA_TRANS_MODE_CODERATE_K_DL, NULL,
&ParaValueInt, NULL);
  Error = WinProp_Net_TransmissionMode_Para_Set(
   ProjectHandle, CurrentService, NET_PARA_TRANS_MODE_CODERATE_K_UL, NULL,
&ParaValueInt, NULL);

  ParaValueInt = TransmissionMode_Coderate_N[CurrentService];
  Error = WinProp_Net_TransmissionMode_Para_Set(
   ProjectHandle, CurrentService, NET_PARA_TRANS_MODE_CODERATE_N_DL, NULL,
&ParaValueInt, NULL);
  Error = WinProp_Net_TransmissionMode_Para_Set(
   ProjectHandle, CurrentService, NET_PARA_TRANS_MODE_CODERATE_N_UL, NULL,
&ParaValueInt, NULL);

  // Set number of resource blocks.
  ParaValueInt = TransmissionMode_Resource_Blocks_DL[CurrentService];
  Error = WinProp_Net_TransmissionMode_Para_Set(
   ProjectHandle, CurrentService, NET_PARA_TRANS_MODE_RESOURCE_BLOCKS_DL, NULL,
&ParaValueInt, NULL);
  ParaValueInt = TransmissionMode_Resource_Blocks_UL[CurrentService];
  Error = WinProp_Net_TransmissionMode_Para_Set(
   ProjectHandle, CurrentService, NET_PARA_TRANS_MODE_RESOURCE_BLOCKS_UL, NULL,
&ParaValueInt, NULL);

  // Set overhead ratio.
  ParaValueDouble = TransmissionMode_Overhead_Ratio_UL[CurrentService];
  Error = WinProp_Net_TransmissionMode_Para_Set(
   ProjectHandle, CurrentService, NET_PARA_TRANS_MODE_OVERHEAD_RATIO_UL,
&ParaValueDouble, NULL, NULL);
  ParaValueDouble = TransmissionMode_Overhead_Ratio_DL[CurrentService];
  Error = WinProp_Net_TransmissionMode_Para_Set(
   ProjectHandle, CurrentService, NET_PARA_TRANS_MODE_OVERHEAD_RATIO_DL,
&ParaValueDouble, NULL, NULL);

  // Set required SNIR.
  ParaValueDouble = TransmissionMode_SNIR[CurrentService];
  Error = WinProp_Net_TransmissionMode_Para_Set(
   ProjectHandle, CurrentService, NET_PARA_TRANS_MODE_SNIR_DL, &ParaValueDouble,
NULL, NULL);
  Error = WinProp_Net_TransmissionMode_Para_Set(
```

```
    ProjectHandle, CurrentService, NET_PARA_TRANS_MODE_SNIR_UL, &ParaValueDouble,
NULL, NULL);

   // Set modulation scheme.
   ParaValueInt = TransmissionMode_MCS[CurrentService];
   Error = WinProp_Net_TransmissionMode_Para_Set(
    ProjectHandle, CurrentService, NET_PARA_TRANS_MODE_MODULATION_UL, NULL,
&ParaValueInt, NULL);
   Error = WinProp_Net_TransmissionMode_Para_Set(
    ProjectHandle, CurrentService, NET_PARA_TRANS_MODE_MODULATION_DL, NULL,
&ParaValueInt, NULL);

   // Set power backoff.
   ParaValueDouble = TransmissionMode_Backoff[CurrentService];
   Error = WinProp_Net_TransmissionMode_Para_Set(
    ProjectHandle, CurrentService, NET_PARA_TRANS_MODE_POWER_BACKOFF_UL,
&ParaValueDouble, NULL, NULL);
   Error = WinProp_Net_TransmissionMode_Para_Set(
    ProjectHandle, CurrentService, NET_PARA_TRANS_MODE_POWER_BACKOFF_DL,
&ParaValueDouble, NULL, NULL);

   // Get bitrate in kBit/s for current transmission mode. Bitrate is automatically
computed based
   // on parameters previously set and general air interface parameters.
   Error = WinProp_Net_TransmissionMode_Para_Get(
    ProjectHandle, CurrentService, NET_PARA_TRANS_MODE_BITRATE_DL, &ParaValueDouble,
NULL, NULL);
   TransmissionMode_Bitrate[CurrentService] = ParaValueDouble;
  }
 }

 // Check if number of services is correct.
 if (Error == 0)
 {
  int NrServices = 0;
  Error = WinProp_Net_Project_Para_Get(ProjectHandle, NET_PARA_SERVICES, NULL,
&NrServices, NULL);
  if (Error == 0)
  {
   if (NrServices != SERVICES_5G)
    Error = 1;
  }
 }

 // -------------------------  Project parameters  ------------------------- //
 // Set MS maximum power
 if (Error == 0)
 {
  ParaValueDouble = 23.; // Power in dBm
  Error = WinProp_Net_Project_Para_Set(ProjectHandle, NET_PARA_MS_MAX_TX_POWER,
&ParaValueDouble, NULL, NULL);
 }

 // Set resolution for result matrix.
 if (Error == 0)
 {
  ParaValueDouble = GeneralParameters.Resolution;
  Error = WinProp_Net_Project_Para_Set(ProjectHandle, NET_PARA_RESOLUTION,
&ParaValueDouble, NULL, NULL);
 }
 // set the simulation area
 if (Error == 0)
 {
```

```
 ParaValueDouble = GeneralParameters.UrbanLowerLeftX;
 Error = WinProp_Net_Project_Para_Set(ProjectHandle, NET_PARA_AREA_LL_X,
&ParaValueDouble, NULL, NULL);
}
if (Error == 0)
{
 ParaValueDouble = GeneralParameters.UrbanLowerLeftY;
 Error = WinProp_Net_Project_Para_Set(ProjectHandle, NET_PARA_AREA_LL_Y,
&ParaValueDouble, NULL, NULL);
}
if (Error == 0)
{
 ParaValueDouble = GeneralParameters.UrbanUpperRightX;
 Error = WinProp_Net_Project_Para_Set(ProjectHandle, NET_PARA_AREA_UR_X,
&ParaValueDouble, NULL, NULL);
}
if (Error == 0)
{
 ParaValueDouble = GeneralParameters.UrbanUpperRightY;
 Error = WinProp_Net_Project_Para_Set(ProjectHandle, NET_PARA_AREA_UR_Y,
&ParaValueDouble, NULL, NULL);
}

// Set paths for additional output in WinProp file format.
if (Error == 0)
{
 Error = WinProp_Net_Project_Para_Set(ProjectHandle, NET_PARA_OUTPUT_WINPROP, NULL,
NULL, API_DATA_FOLDER "output/monte_carlo_5g");
}

// ------------------------  Cell assignment   ------------------------------ //

// The default cell assignment for the selected air interface (i.e.
NET_AIRINTERFACE_5G_FDD_GENERIC)
//  is highest power for all carries, definition of min. required SNIR and power

// Set Min required SNIR
if (Error == 0)
{
 ParaValueDouble = -3.;
 Error = WinProp_Net_Project_Para_Set(ProjectHandle,
NET_PARA_CELL_ASSIGNMENT_MIN_REQ_SNIR, &ParaValueDouble, NULL, NULL);
}

// Set Min required power
if (Error == 0)
{
 ParaValueDouble = -95.;
 Error = WinProp_Net_Project_Para_Set(ProjectHandle,
NET_PARA_CELL_ASSIGNMENT_MIN_REQ_POWER, &ParaValueDouble, NULL, NULL);
}

// -------------------  numerology and  masking mode  ----------------------- //
// Activate antenna masking for network planning results (propagation results with
omni-pattern)
if (Error == 0)
{
 ParaValueInt = 1;
 Error = WinProp_Net_Project_Para_Set(ProjectHandle, NET_PARA_ANTENNA_MASKING_MODE,
NULL, &ParaValueInt, NULL);
}
```

```
// Set numerology and bandwidth
if (Error == 0)
{
 ParaValueInt = NET_PARA_5G_NUMEROLOGY0_20MHZ_FR1;
 Error = WinProp_Net_Project_Para_Set(ProjectHandle, NET_PARA_BANDWIDTH_MODE_5G,
NULL, &ParaValueInt, NULL);
}

// ----------------------  OFDM parameters  ------------------------------- //
// Set control sub-carriers
if (Error == 0)
{
 ParaValueInt = 1000;
 Error = WinProp_Net_Project_Para_Set(ProjectHandle, NET_PARA_SUBCARRIERS_CONTROL,
NULL, &ParaValueInt, NULL);
}

// Set reference sub-carriers
if (Error == 0)
{
 ParaValueInt = 200;
 Error = WinProp_Net_Project_Para_Set(ProjectHandle, NET_PARA_SUBCARRIERS_REFERENCE,
NULL, &ParaValueInt, NULL);
}

// Set Guard sub-carriers
if (Error == 0)
{
 ParaValueInt = 31;
 Error = WinProp_Net_Project_Para_Set(ProjectHandle, NET_PARA_SUBCARRIERS_GUARD,
NULL, &ParaValueInt, NULL);
}

// Set data symbols
if (Error == 0)
{
 ParaValueInt = 10;
 Error = WinProp_Net_Project_Para_Set(ProjectHandle, NET_PARA_SYMBOLS_DATA, nullptr,
&ParaValueInt, nullptr);
}

// Set control symbols
if (Error == 0)
{
 ParaValueInt = 1;
 Error = WinProp_Net_Project_Para_Set(ProjectHandle, NET_PARA_SYMBOLS_CONTROL, NULL,
&ParaValueInt, NULL);
}

// Set reference symbols
if (Error == 0)
{
 ParaValueInt = 3;
 Error = WinProp_Net_Project_Para_Set(ProjectHandle, NET_PARA_SYMBOLS_REFERENCE,
NULL, &ParaValueInt, NULL);
}

// Set FFT order
if (Error == 0)
{
 ParaValueInt = 2048;
```

```
 Error = WinProp_Net_Project_Para_Set(ProjectHandle, NET_PARA_FFT_SIZE, NULL,
&ParaValueInt, NULL);
}

// ----------------------  Monte Carlo Parameters  -------------------------- //
// Set traffic mode to Monte Carlo
if (Error == 0)
{
 ParaValueInt = NET_TRAFFIC_MODE_STATIC_MONTE_CARLO;
 Error = WinProp_Net_Project_Para_Set(ProjectHandle, NET_PARA_TRAFFIC_MODE, nullptr,
&ParaValueInt, nullptr);
}

// Set Monte Carlo max loops
if (Error == 0)
{
 ParaValueInt = 5;
 Error = WinProp_Net_Project_Para_Set(ProjectHandle,
NET_PARA_TRAFFIC_MONTE_CARLO_LOOPS, nullptr, &ParaValueInt, nullptr);
}

// Set users distributed per square kilometer
if (Error == 0)
{
 ParaValueInt = NET_APPLICATION_AREA_MODE_KILOMETER;
 Error = WinProp_Net_Project_Para_Set(ProjectHandle, NET_PARA_APPLICATION_AREA_MODE,
nullptr, &ParaValueInt, nullptr);
}

// Consider priority of an application in a cell
if (Error == 0)
{
 ParaValueInt = 1;
 Error = WinProp_Net_Project_Para_Set(ProjectHandle,
NET_PARA_TRAFFIC_CELL_SELECT_CONSIDER_PRIO, nullptr, &ParaValueInt, nullptr);
}

//  Number of iterations for traffic simulations
if (Error == 0)
{
 ParaValueInt = 20;
 Error = WinProp_Net_Project_Para_Set(ProjectHandle,
NET_PARA_TRAFFIC_MAX_ITERATIONS, nullptr, &ParaValueInt, nullptr);
}


// Add clutter from the database file
if (Error == 0)
{
 char ClutterTraffic[200];
 sprintf(ClutterTraffic, "%s.odb", my_odb_database);
 Error = WinProp_Net_Clutter_Set(ProjectHandle, CLUTTERMODE_FILE,
ClutterTraffic, false, "", nullptr);
}

// Add applications
if (Error == 0)
{
 int   ApplID = 0;
 int   ApplPriority = 0;
 float ApplActivity = 1.f;
 int   ApplTrafficMode = NET_APPLICATION_TRAFFIC_MODE_OFFERED_TRAFFIC;
```

```
 const char* ApplName = "Browsing";
 Error = WinProp_Net_Application_Add(ProjectHandle, ApplID, ApplPriority,
ApplActivity, ApplTrafficMode, ApplName);

 // add traffic clutter classes to the application
 if (Error == 0)
 {
  int       ClutterClassID = 1;        // ID of clutter class  1 = corresponds to
OUTDOOR clutter as defined in the data base
  float     ArrivalRate    = 0.f;      // Arrival rate in 1/(km^2/s) (not relevant
for NET_APPLICATION_TRAFFIC_MODE_OFFERED_TRAFFIC)
  float     HoldTime       = 0.f;      // Hold Time in [sec] (not relevant for
NET_APPLICATION_TRAFFIC_MODE_OFFERED_TRAFFIC)
  float     OfferedTraffic = 50.f;     // Offered Traffic in Erl/km^2
  WinProp_Net_Application_ClutterClass_Link(ProjectHandle, ApplID, ClutterClassID,
ArrivalRate, HoldTime, OfferedTraffic);
 }

 // add transmission modes to the application
 if (Error == 0)
 {
  int TransModeID = 5; // corresponds to "18 - 64 QAM - R 822" as defined above.
  Error = WinProp_Net_Application_TransmissionMode_Link(ProjectHandle, ApplID,
TransModeID, 0);
 }
}

// Callback functions.
   WinProp_Callback Callback;
   WinProp_Structure_Init_Callback(&Callback);
Callback.Percentage = CallbackProgress;
Callback.Message = CallbackMessage;
Callback.Error = CallbackError;

//-------------------------------------------------------------------------
   // Compute wave propagation for three cells
   //-------------------------------------------------------------------------
for (int cell = 0; cell < MAX_CELLS; cell++)
{
 /*---------------------- Compute outdoor prediction -----------------------*/
 Error = OutdoorPlugIn_ComputePrediction(
  &Antenna[cell], &GeneralParameters, NULL, 0, NULL, NULL, NULL, NULL, &Callback,
&PropResults[cell], NULL, NULL, NULL);

 /*----------------------- Do something with results -----------------------*/
 if (Error == 0)
 {
  char PowerOutputFile[1200];
  sprintf(PowerOutputFile, "%s%s%s%s", API_DATA_FOLDER, "output/monte_carlo_5g/",
Antenna[cell].Name,".txt");

  write_ascii(&PropResults[cell], PowerOutputFile);
 }
 else
 {
  /* Error during prediction. Print error message. */
  CallbackError(GeneralParameters.ErrorMessageMain, Error);
 }
}

// Add all propagation maps which have been computed.
```

```
// Set the antenna pattern for masking the already calculated omni-propagation
results
if (Error == 0)
{
 int MapIndex[MAX_CELLS];
 for (int Count = 0; Count < MAX_CELLS; Count++)
 {
  if (Error == 0)
  {
   // Masking mode -->  set antenna pattens for network planning results
   Antenna[Count].Pattern = &WinPropPattern;

   Error = WinProp_Net_PropagationMap_Add(
    ProjectHandle, &MapIndex[Count], &Antenna[Count], &PropResults[Count]);
  }
 }
}

// Desired network planning results in WinProp result format
Error = WinProp_Net_Project_Result_Switch(ProjectHandle, NET_RESULT_BEST_SERVER, 1);
Error = WinProp_Net_Project_Result_Switch(ProjectHandle, NET_RESULT_MAX_DATARATE,
1);
Error = WinProp_Net_Project_Result_Switch(ProjectHandle, NET_RESULT_MAX_REC_POWER,
1);
Error = WinProp_Net_Project_Result_Switch(ProjectHandle, NET_RESULT_REC_POWER, 1);
Error = WinProp_Net_Project_Result_Switch(ProjectHandle, NET_RESULT_SNIR, 1);
Error = WinProp_Net_Project_Result_Switch(ProjectHandle, NET_RESULT_TRAFFIC_OFFERED,
1);
Error = WinProp_Net_Project_Result_Switch(ProjectHandle, NET_RESULT_TRAFFIC_SERVED,
1);
Error = WinProp_Net_Project_Result_Switch(ProjectHandle, NET_RESULT_TRAFFIC_BLOCKED,
1);
Error = WinProp_Net_Project_Result_Switch(ProjectHandle, NET_RESULT_TRAFFIC_STATE,
1);

//-----------------------------------------------------------------------------
   //  Compute network planning results
   //-----------------------------------------------------------------------------
if (Error == 0)
{
 char HeightString[200];
 sprintf(HeightString, "%s", "");

 // Generate string with height values, e.g. a string like "1.5 2.5 3.5".
 for (int Height = 0; Height < NrPredictionHeights; Height++)
 {
  // Add current height to string.
  char thisHeightStr[50];
  sprintf(thisHeightStr, "%.2f ", PredictionHeights[Height]);
  strcat(HeightString, thisHeightStr);
 }

 // Send heights to WinProp API.
 Error = WinProp_Net_Project_Para_Set(ProjectHandle, NET_PARA_HEIGHT_MULTIPLE, NULL,
NULL, HeightString);

 // Start network computation.
 if (Error == 0)
 {
  Error = WinProp_Net_Project_Compute(ProjectHandle, &Callback);
 }
}
```

```
 // ------------------------------------------------------------------------
 // Retrieve Monte-Carlo statistics
 // ------------------------------------------------------------------------
 if (Error == 0)
 {
  WinProp_ResultMonteCarlo* ResultMonteCarlo = NULL;
  Error = WinProp_Net_NetworkMonteCarloStats_Get(ProjectHandle, &ResultMonteCarlo);

  if (Error == 0)
  {
   char NameForOutput[1200];
   sprintf(NameForOutput, "%s%s", API_DATA_FOLDER, "output/monte_carlo_5g/
MonteCarloStatistics.txt");
   write_ascii_monte_carlo_statistic(ResultMonteCarlo, NameForOutput);

  }
 }

 // ------------------------------------------------------------------------
 // Retrieve some other results
 // ------------------------------------------------------------------------
 // As an example: retrieve max. throughput (kbps) per pixel.
 if (Error == 0)
 {
  WinProp_Result* MaxThroughput = NULL;
  Error = WinProp_Net_NetworkMap_Get(ProjectHandle, -1, NET_RESULT_MAX_THROUGHPUT,
 &MaxThroughput);

  // Write max. throughput result to ASCII file.
  if (Error == 0)
  {
   char NameForOutput[1200];
   sprintf(NameForOutput, "%s%s", API_DATA_FOLDER, "output/monte_carlo_5g/Max
 Throughput DownLink.txt");
   write_ascii(MaxThroughput, NameForOutput);
  }

 }

 // As another example: retrieve max. received power (dBm) per pixel.
 if (Error == 0)
 {
  WinProp_Result* MaxPower = NULL;
  Error = WinProp_Net_NetworkMap_Get(ProjectHandle, -1, NET_RESULT_MAX_REC_POWER,
 &MaxPower);

  // Write max. throughput result to ASCII file.
  if (Error == 0)
  {
   char NameForOutput[1200];
   sprintf(NameForOutput, "%s%s", API_DATA_FOLDER, "output/monte_carlo_5g/Max
 Received Power.txt");
   write_ascii(MaxPower, NameForOutput);
  }

 }


 // As another example: retrieve SNIR (dB) per pixel.
 if (Error == 0)
 {
```

```
  WinProp_Result* MaxSNIR = NULL;
  Error = WinProp_Net_NetworkMap_Get(ProjectHandle, -1, NET_RESULT_SNIR, &MaxSNIR);

  // Write max. throughput result to ASCII file.
  if (Error == 0)
  {
   char NameForOutput[1200];
   sprintf(NameForOutput, "%s%s", API_DATA_FOLDER, "output/monte_carlo_5g/Max
SNIR.txt");
   write_ascii(MaxSNIR, NameForOutput);
  }
 }

 // ----------------------------------------------------------------------------
 // Free memory
 // ----------------------------------------------------------------------------
 for (int Count = 0; Count < MAX_CELLS; Count++)
 {
  // Free propagation results.
  WinProp_FreeResult(&PropResults[Count]);
 }

 // Close network project.
 Error = WinProp_Net_Project_Close(ProjectHandle);

 return 0;
}

int _STD_CALL CallbackMessage(const char * Text)
{
 if (Text == nullptr)
  return 0;

 std::cout << "\n" << Text;

 return(0);
}

int _STD_CALL CallbackError(const char * Text, int Error)
{
 if (Text == nullptr)
  return 0;

 std::cout << "\n";

#ifdef __LINUX
 std::cout << "\033[31m" << "Error (" << Error << "): "; // highlight error in red
 color
#else
 HANDLE hConsole = GetStdHandle(STD_OUTPUT_HANDLE);
 SetConsoleTextAttribute(hConsole, FOREGROUND_RED);
 std::cout << "Error (" << Error << "): ";
#endif // __LINUX
 std::cout << Text;

#ifdef __LINUX
 std::cout << "\033[0m"; // highlight error in red color
#else
 SetConsoleTextAttribute(hConsole, FOREGROUND_RED | FOREGROUND_BLUE |
 FOREGROUND_GREEN);
#endif // __LINUX
```

```
 return 0;
}

int _STD_CALL CallbackProgress(int value, const char* text)
{
 char Line[200];

 sprintf(Line, "\n%d%% %s", value, text);
 std::cout << Line;

 return(0);
}

// Helper functions
void write_ascii(const WinProp_Result *Resultmatrix, const char* Filename) {

 if (Resultmatrix)
 {
  FILE* OutputFile = fopen(Filename, "w");
  if (OutputFile)
  {
   /* Loop through WinPropall pixels. */
   for (int x = 0; x < Resultmatrix->Columns; x++)
   {
    for (int y = 0; y < Resultmatrix->Lines; y++)
    {
     /* Compute real coordinates. */
     double Coordinate_X = Resultmatrix->LowerLeftX + ((double)x + 0.5) *
 Resultmatrix->Resolution;
     double Coordinate_Y = Resultmatrix->LowerLeftY + ((double)y + 0.5) *
 Resultmatrix->Resolution;

     /* Check if pixel was computed or not */
     if (Resultmatrix->Matrix[0][x][y] > -1000)
     {
      fprintf(OutputFile, "%.5f\t%.5f\t%.4f\n", Coordinate_X, Coordinate_Y,
 Resultmatrix->Matrix[0][x][y]);
     }
     else
     {
      fprintf(OutputFile, "%.5f\t%.5f\t%s\n", Coordinate_X, Coordinate_Y, "N.C.");
     }
    }
   }

   /* Close file. */
   fclose(OutputFile);
  }
  else
  {
   printf("\nCould not open the File: %s for writing.\n", Filename);
  }
 }
}

void CarrierPropertiesSet(WinProp_Antenna *Antenna, const WinProp_Carrier *Carrier) {

 Antenna->Carriers.CarrierID = Carrier->CarrierID;
 Antenna->Carriers.SystemID = Carrier->SystemID;
 Antenna->Carriers.CellLoad = Carrier->CellLoad;
 Antenna->Carriers.MimoID = Carrier->MimoID;
```

```
 Antenna->Carriers.NoiseRiseUL = Carrier->NoiseRiseUL;
 Antenna->Carriers.PowerBackoffPilotDL = Carrier->PowerBackoffPilotDL;

 // 5G Carrier parameters
 Antenna->Carriers.Numerology = Carrier->Numerology;
 Antenna->Carriers.SymbolsPerSlot = Carrier->SymbolsPerSlot;
 Antenna->Carriers.NrAntennaBeams = Carrier->NrAntennaBeams;
 Antenna->Carriers.BeamGainCtrlServer = Carrier->BeamGainCtrlServer;
 Antenna->Carriers.BeamGainCtrlInterferer = Carrier->BeamGainCtrlInterferer;
 Antenna->Carriers.BeamGainDataServer = Carrier->BeamGainDataServer;
 Antenna->Carriers.BeamGainDataInterferer = Carrier->BeamGainDataInterferer;
 Antenna->Carriers.TDD_Slots_DL = Carrier->TDD_Slots_DL;
 Antenna->Carriers.TDD_Slots_UL = Carrier->TDD_Slots_UL;
 Antenna->Carriers.TDD_Slots_Flex = Carrier->TDD_Slots_Flex;
}

/**
 * Writes Monte Carlo statistics in an ASCII file
 *
 * @param [in,out] ResultMonteCarlo If non-null, the result to be written.
 * @param [in,out] Filename   If non-null, filename of the file.
 */

void write_ascii_monte_carlo_statistic(
 const WinProp_ResultMonteCarlo* ResultMonteCarlo,
 const char* Filename)
{
 if (ResultMonteCarlo == NULL)
  return;

 FILE* Outfile = fopen(Filename, "w+");

 if (Outfile)
 {
  char         sepChar = ' ';
  char         format1[100];
  char         format2[100];
  char         text[400];


 fprintf(Outfile, "%s", "-----------------------------------------------------------------------
\n");
  fprintf(Outfile, "Number of evaluated snapshots:%c%d\n", sepChar, ResultMonteCarlo-
>nbrSnapshots);
  fprintf(Outfile, "Number of evaluated cells:%c%d\n", sepChar, ResultMonteCarlo-
>nbrCells);
  fprintf(Outfile, "Number of evaluated applications:%c%d\n", sepChar,
 ResultMonteCarlo->nbrApplications);

 fprintf(Outfile, "%s", "-----------------------------------------------------------------------
\n");
  fprintf(Outfile, "%s", "\n");

  /* assign width of columns */
  sprintf(format1, "%%%ds", 15);
  sprintf(format2, "%%%ds", 14);

  /* loop over all cells (index -1 is for total) */
  for (int cell = -1; cell < (int)ResultMonteCarlo->nbrCells; cell++)
  {
   /* write header of table */
   fprintf(Outfile, "%s", "*********************************\n");
```

```
   if (cell >= 0)
   {
    fprintf(Outfile, "***%cCell %d", sepChar, (cell + 1));
    fprintf(Outfile, "%s", "\n*********************************");
   }
   else
   {
    sprintf(text, "***%cTotal simulation area", sepChar);
    fprintf(Outfile, "%-30s", text);
    fprintf(Outfile, "%s", "\n*********************************");
   }

   /* Write snapshots */
   for (int snapshot = (int)ResultMonteCarlo->nbrSnapshots; snapshot >= 0;
 snapshot--)
   {
    if (snapshot == (int)ResultMonteCarlo->nbrSnapshots)
     sprintf(text, "%cAverage", sepChar);
    else
     sprintf(text, "%cSnapshot %d", sepChar, (snapshot + 1));
    if (snapshot == (int)ResultMonteCarlo->nbrSnapshots)
     fprintf(Outfile, format2, text);
    else
     fprintf(Outfile, format1, text);
   }
   fprintf(Outfile, "%s", "\n");


   /* write cell load (Tx Power Downlink) */
   if (cell >= 0)
   {
    fprintf(Outfile, "%-31s", "    Tx Power Load (Downlink):    ");
    for (int snapshot = (int)ResultMonteCarlo->nbrSnapshots; snapshot >= 0;
 snapshot--)
    {
     // average value of tx power
     if (snapshot == (int)ResultMonteCarlo->nbrSnapshots)
     {
      float average = 0.f;
      for (unsigned int s = 0; s < ResultMonteCarlo->nbrSnapshots; s++)
      {
       average += ResultMonteCarlo->trxLoad[s][cell].LoadPowerDL;
      }

      if (ResultMonteCarlo->nbrSnapshots > 0)
      {
       average /= ResultMonteCarlo->nbrSnapshots;
      }
      sprintf(text, "%c%.1f%c%%", sepChar, average * 100.f, sepChar);
      fprintf(Outfile, format2, text);
     }
     else
     {
      // values for each snapshot
      sprintf(text, "%c%.1f%c%%", sepChar, ResultMonteCarlo->trxLoad[snapshot]
[cell].LoadPowerDL * 100.f, sepChar);
      fprintf(Outfile, format1, text);
     }
    }
    fprintf(Outfile, "%s", "\n");
   }
```

```
   /* write noise rise (uplink) */
   if (cell >= 0)
   {
    fprintf(Outfile, "%-33s", "    Noise Rise (Uplink):        ");
    for (int snapshot = (int)ResultMonteCarlo->nbrSnapshots; snapshot >= 0;
 snapshot--)
    {
     // average value of noise rise
     if (snapshot == (int)ResultMonteCarlo->nbrSnapshots)
     {
      float average = 0.f;
      for (unsigned int s = 0; s < ResultMonteCarlo->nbrSnapshots; s++)
      {
       average += ResultMonteCarlo->trxLoad[s][cell].NoiseRiseUL;
      }

      if (ResultMonteCarlo->nbrSnapshots > 0)
      {
       average /= ResultMonteCarlo->nbrSnapshots;
      }
      sprintf(text, "%c%.1f%cdB", sepChar, average, sepChar);
      fprintf(Outfile, format2, text);
     }
     else
     {
      // values for each snapshot
      sprintf(text, "%c%.1f%cdB", sepChar, ResultMonteCarlo->trxLoad[snapshot]
[cell].NoiseRiseUL, sepChar);
      fprintf(Outfile, format1, text);
     }
    }
    fprintf(Outfile, "%s", "\n");
   }

   /* write resources load (downlink) */
   if (cell >= 0)
   {
    fprintf(Outfile, "%-32s", "    Resources Load (Downlink): ");
    for (int snapshot = (int)ResultMonteCarlo->nbrSnapshots; snapshot >= 0;
 snapshot--)
    {
     // average value of DL resources load
     if (snapshot == (int)ResultMonteCarlo->nbrSnapshots)
     {
      float average = 0.f;
      for (unsigned int s = 0; s < ResultMonteCarlo->nbrSnapshots; s++)
      {
       average += ResultMonteCarlo->trxLoad[s][cell].LoadResourcesDL;
      }

      if (ResultMonteCarlo->nbrSnapshots > 0)
      {
       average /= ResultMonteCarlo->nbrSnapshots;
      }
      sprintf(text, "%c%.1f%c%%", sepChar, average * 100.f, sepChar);
      fprintf(Outfile, format2, text);
     }
     else
     {
      // values for each snapshot
      sprintf(text, "%c%.1f%c%%", sepChar, ResultMonteCarlo->trxLoad[snapshot]
[cell].LoadResourcesDL * 100.f, sepChar);
```

```
      fprintf(Outfile, format1, text);
     }
    }
    fprintf(Outfile, "%s", "\n");
   }

   /* write resources load (downlink) */
   if (cell >= 0)
   {
    fprintf(Outfile, "%-32s", "    Resources Load (Uplink): ");
    for (int snapshot = (int)ResultMonteCarlo->nbrSnapshots; snapshot >= 0;
 snapshot--)
    {
     // average value of UL resources load
     if (snapshot == (int)ResultMonteCarlo->nbrSnapshots)
     {
      float average = 0.f;
      for (unsigned int s = 0; s < ResultMonteCarlo->nbrSnapshots; s++)
      {
       average += ResultMonteCarlo->trxLoad[s][cell].LoadResourcesUL;
      }

      if (ResultMonteCarlo->nbrSnapshots > 0)
      {
       average /= ResultMonteCarlo->nbrSnapshots;
      }
      sprintf(text, "%c%.1f%c%%", sepChar, average * 100.f, sepChar);
      fprintf(Outfile, format2, text);
     }
     else
     {
      // values for each snapshot
      sprintf(text, "%c%.1f%c%%", sepChar, ResultMonteCarlo->trxLoad[snapshot]
[cell].LoadResourcesUL * 100.f, sepChar);
      fprintf(Outfile, format1, text);
     }
    }
    fprintf(Outfile, "%s", "\n");
   }

   /* write throughput (downlink) */
   if (cell >= 0)
   {
    fprintf(Outfile, "%-33s", "    Throughput (Downlink):       ");
    for (int snapshot = (int)ResultMonteCarlo->nbrSnapshots; snapshot >= 0;
 snapshot--)
    {
     float ThroughputDL = 0.f;
     if (snapshot == (int)ResultMonteCarlo->nbrSnapshots)
     {
      float average = 0.f;
      for (unsigned int s = 0; s < ResultMonteCarlo->nbrSnapshots; s++)
      {
       average += ResultMonteCarlo->trxLoad[s][cell].ThroughputUL;
      }
      if (ResultMonteCarlo->nbrSnapshots > 0)
      {
       average /= ResultMonteCarlo->nbrSnapshots;
      }
      // average throughput
      ThroughputDL = average;
     }
```

```
     else
     {
      // values for each snapshot
      ThroughputDL = ResultMonteCarlo->trxLoad[snapshot][cell].ThroughputUL;
     }

     // format throughput
     if (ThroughputDL > 1e9)
      sprintf(text, "%c%.1f%cGB/s", sepChar, ThroughputDL / 1e9, sepChar);
     else if (ThroughputDL > 1e6)
      sprintf(text, "%c%.1f%cMB/s", sepChar, ThroughputDL / 1e6, sepChar);
     else if (ThroughputDL > 1e3)
      sprintf(text, "%c%.1f%ckB/s", sepChar, ThroughputDL / 1e3, sepChar);
     else
      sprintf(text, "%c%.1f%cB/s", sepChar, ThroughputDL, sepChar);

     if (snapshot == (int)ResultMonteCarlo->nbrSnapshots)
      fprintf(Outfile, format2, text);
     else
      fprintf(Outfile, format1, text);
    }
    fprintf(Outfile, "%s", "\n");
   }

   /* write throughput (uplink) */
   if (cell >= 0)
   {
    fprintf(Outfile, "%-33s", "    Throughput (uplink):      ");
    for (int snapshot = (int)ResultMonteCarlo->nbrSnapshots; snapshot >= 0;
snapshot--)
    {
     float ThroughputUL = 0.f;
     if (snapshot == (int)ResultMonteCarlo->nbrSnapshots)
     {
      float average = 0.f;
      for (unsigned int s = 0; s < ResultMonteCarlo->nbrSnapshots; s++)
      {
       average += ResultMonteCarlo->trxLoad[s][cell].ThroughputUL;
      }
      if (ResultMonteCarlo->nbrSnapshots > 0)
      {
       average /= ResultMonteCarlo->nbrSnapshots;
      }
      // average throughput
      ThroughputUL = average;
     }
     else
     {
      // values for each snapshot
      ThroughputUL = ResultMonteCarlo->trxLoad[snapshot][cell].ThroughputUL;
     }

     // format throughput
     if (ThroughputUL > 1e9)
      sprintf(text, "%c%.1f%cGB/s", sepChar, ThroughputUL / 1e9, sepChar);
     else if (ThroughputUL > 1e6)
      sprintf(text, "%c%.1f%cMB/s", sepChar, ThroughputUL / 1e6, sepChar);
     else if (ThroughputUL > 1e3)
      sprintf(text, "%c%.1f%ckB/s", sepChar, ThroughputUL / 1e3, sepChar);
     else
      sprintf(text, "%c%.1f%cB/s", sepChar, ThroughputUL, sepChar);
```

```
     if (snapshot == (int)ResultMonteCarlo->nbrSnapshots)
      fprintf(Outfile, format2, text);
     else
      fprintf(Outfile, format1, text);
    }
    fprintf(Outfile, "%s", "\n");
   }

   /* write number of served, blocked, and not assigned mobiles for each application
*/
   for (unsigned int app = 0; app <= ResultMonteCarlo->nbrApplications; app++)
   {
    /* write name of application */
    if (app < ResultMonteCarlo->nbrApplications)
     sprintf(text, "%s %d", "\n Applications id: ", app);
    else
     sprintf(text, "%s", "\nAll Applications");

    fprintf(Outfile, "%-20s\n", text);

    for (int MobileType = 0; MobileType < 4; MobileType++)
    {
     // Mobiles served  = 0
     // Mobiles blocked(traffic) = 1
     // Mobiles blocked(quality) = 2
     // Mobiles not assigned = 3
     switch (MobileType)
     {
     case 0:
      fprintf(Outfile, "   %cMobiles served:           ", sepChar);
      break;
     case 1:
      fprintf(Outfile, "   %cMobiles blocked (traffic):", sepChar);
      break;
     case 2:
      fprintf(Outfile, "   %cMobiles blocked (quality):", sepChar);
      break;
     case 3:
      fprintf(Outfile, "   %cMobiles not assigned:     ", sepChar);
      break;
     default:
      break;
     }

     for (int snapshot = (int)ResultMonteCarlo->nbrSnapshots; snapshot >= 0;
snapshot--)
     {
      float nbr = 0.f;

      // All Applications case
      if (app == ResultMonteCarlo->nbrApplications)
      {
       for (unsigned int a = 0; a < ResultMonteCarlo->nbrApplications; a++)
       {
        // specific cell
        if (cell >= 0)
        {
         if (snapshot == (int)ResultMonteCarlo->nbrSnapshots)
         {
          // average number (over snapshots) for all apps  for a specific cell
          for (unsigned int s = 0; s < ResultMonteCarlo->nbrSnapshots; s++)
          {
```

```
            switch (MobileType)
            {
            case 0:
             nbr += ResultMonteCarlo->statisticCell[s][cell][a].nbrServed;
             break;
            case 1:
             nbr += ResultMonteCarlo->statisticCell[s][cell][a].nbrBlocked;
             break;
            case 2:
             nbr += ResultMonteCarlo->statisticCell[s][cell][a].nbrNotConnected;
             break;
            case 3:
             nbr += ResultMonteCarlo->statisticCell[s][cell][a].nbrNotAssigned;
             break;
            default:
             break;
            }
           }
          }
          else
          {
           // number for all apps for a specific cell and a specific snapshot
           switch (MobileType)
           {
           case 0:
            nbr += ResultMonteCarlo->statisticCell[snapshot][cell][a].nbrServed;
            break;
           case 1:
            nbr += ResultMonteCarlo->statisticCell[snapshot][cell][a].nbrBlocked;
            break;
           case 2:
            nbr += ResultMonteCarlo->statisticCell[snapshot][cell][a].nbrNotConnected;
            break;
           case 3:
            nbr += ResultMonteCarlo->statisticCell[snapshot][cell][a].nbrNotAssigned;
            break;
           default:
            break;
           }
          }

         }
         else // all cells
         {
          // average number (over snapshots) for all apps for all cells
          if (snapshot == (int)ResultMonteCarlo->nbrSnapshots)
          {
           for (unsigned int s = 0; s < ResultMonteCarlo->nbrSnapshots; s++)
           {
            switch (MobileType)
            {
            case 0:
             nbr += ResultMonteCarlo->statisticTotal[s][a].nbrServed;
             break;
            case 1:
             nbr += ResultMonteCarlo->statisticTotal[s][a].nbrBlocked;
             break;
            case 2:
             nbr += ResultMonteCarlo->statisticTotal[s][a].nbrNotConnected;
             break;
            case 3:
```

```
            nbr += ResultMonteCarlo->statisticTotal[s][a].nbrNotAssigned;
            break;
           default:
            break;
           }
          }
         }
         else
         {
          // number for all apps for all cells per snapshot
          switch (MobileType)
          {
          case 0:
           nbr += ResultMonteCarlo->statisticTotal[snapshot][a].nbrServed;
           break;
          case 1:
           nbr += ResultMonteCarlo->statisticTotal[snapshot][a].nbrBlocked;
           break;
          case 2:
           nbr += ResultMonteCarlo->statisticTotal[snapshot][a].nbrNotConnected;
           break;
          case 3:
           nbr += ResultMonteCarlo->statisticTotal[snapshot][a].nbrNotAssigned;
           break;
          default:
           break;
          }
         }
        }
       }
      else // individual application case
      {
       // specific cell
       if (cell >= 0)
       {
        // average number (over snapshots) for a specific apps and a specific cell
        if (snapshot == (int)ResultMonteCarlo->nbrSnapshots)
        {
         for (unsigned int s = 0; s < ResultMonteCarlo->nbrSnapshots; s++)
         {
          switch (MobileType)
          {
          case 0:
           nbr += ResultMonteCarlo->statisticCell[s][cell][app].nbrServed;
           break;
          case 1:
           nbr += ResultMonteCarlo->statisticCell[s][cell][app].nbrBlocked;
           break;
          case 2:
           nbr += ResultMonteCarlo->statisticCell[s][cell][app].nbrNotConnected;
           break;
          case 3:
           nbr += ResultMonteCarlo->statisticCell[s][cell][app].nbrNotAssigned;
           break;
          default:
           break;
          }
         }
        }
        else
        {
```

```
          // number for a specific app,  a specific cell, and a specific snapshot
          switch (MobileType)
          {
          case 0:
           nbr += ResultMonteCarlo->statisticCell[snapshot][cell][app].nbrServed;
           break;
          case 1:
           nbr += ResultMonteCarlo->statisticCell[snapshot][cell][app].nbrBlocked;
           break;
          case 2:
           nbr += ResultMonteCarlo->statisticCell[snapshot][cell]
[app].nbrNotConnected;
           break;
          case 3:
           nbr += ResultMonteCarlo->statisticCell[snapshot][cell][app].nbrNotAssigned;
           break;
          default:
           break;
          }
         }
        else // all cells

         // average number (over snapshots) for a specific app and for all cells
         if (snapshot == (int)ResultMonteCarlo->nbrSnapshots)
         {
          for (unsigned int s = 0; s < ResultMonteCarlo->nbrSnapshots; s++)
          {
           switch (MobileType)
           {
           case 0:
            nbr += ResultMonteCarlo->statisticTotal[s][app].nbrServed;
            break;
           case 1:
            nbr += ResultMonteCarlo->statisticTotal[s][app].nbrBlocked;
            break;
           case 2:
            nbr += ResultMonteCarlo->statisticTotal[s][app].nbrNotConnected;
            break;
           case 3:
            nbr += ResultMonteCarlo->statisticTotal[s][app].nbrNotAssigned;
            break;
           default:
            break;
           }
          }
         }
         else
         {
          // number  for a specific app, a specif specific and for all cells
          switch (MobileType)
          {
          case 0:
           nbr += ResultMonteCarlo->statisticTotal[snapshot][app].nbrServed;
           break;
          case 1:
           nbr += ResultMonteCarlo->statisticTotal[snapshot][app].nbrBlocked;
           break;
          case 2:
           nbr += ResultMonteCarlo->statisticTotal[snapshot][app].nbrNotConnected;
           break;
```

```
          case 3:
           nbr += ResultMonteCarlo->statisticTotal[snapshot][app].nbrNotAssigned;
           break;
          default:
           break;
          }
         }
       }
       if (snapshot == (int)ResultMonteCarlo->nbrSnapshots)
       {
        double nbrDouble = ((double)nbr) / ((double)ResultMonteCarlo->nbrSnapshots);
        sprintf(text, "%c%.11f", sepChar, nbrDouble);
        fprintf(Outfile, format2, text);
       }
       else
       {
        sprintf(text, "%c%.1f", sepChar, nbr);
        fprintf(Outfile, format1, text);
       }
      }
      fprintf(Outfile, "%s", "\n");
     }
    }
    /* next cell */
    fprintf(Outfile, "%s", "\n\n\n");
   }
   /* close file */
   fclose(Outfile);
  }
  else
  {
   printf("\nCould not open the File: %s for writing.\n", Filename);
  }

}
```

2.6.14 Network planning with 5G TDD

Detailed Description

This is an example of how to use the WinProp API for network planning with the 5G TDD protocol. The full example is distributed with the installation.

```
#include <stdio.h>
#include <string>
#include <iostream>
#include <cstring>

#include "network_planning_5g_tdd.h"

#ifndef API_DATA_FOLDER
#define API_DATA_FOLDER "../../api/winprop/data/"
#endif // !API_DATA_FOLDER
```

```
#define SERVICES_5G 13
#define MAX_CELLS 1

void AntennaPropertiesSet(
    WinProp_Antenna* Antenna,
    double CoordinateX,
    double CoordinateY,
    double Height,
    double Frequency,
    const char* Name,
    double Power,
    WINPROP_POWER_MODE PowerMode,
    WINPROP_MODEL_INDOOR Model,
    double Azimuth,
    double Downtilt,
    int Id)
{
    Antenna->Longitude_X = CoordinateX;
    Antenna->Latitude_Y = CoordinateY;
    Antenna->Height = Height;
    Antenna->Power = Power;
    Antenna->PowerMode = PowerMode;
    Antenna->Frequency = Frequency;
    sprintf(Antenna->Name, "%s", Name);
    Antenna->Model = Model;
    Antenna->Azimuth = Azimuth;
    Antenna->Downtilt = Downtilt;
    Antenna->Id = Id;
}

int main(int argc, char** argv)
{

 int Error = 0;
 int ProjectHandle = 0;
 int ParaValueInt = 0;
 double ParaValueDouble = 0.;

 // ------ General and Propagation and Network Planing Parameters ------ //
 WinProp_ParaMain GeneralParameters;
 WinProp_Pattern  WinPropPattern[MAX_CELLS];
 WinProp_Antenna  Antenna[MAX_CELLS];
 WinProp_Carrier  Carrier[MAX_CELLS];

 // Propagation results
 WinProp_Result   PropResults[MAX_CELLS];

 // Prediction heights
 int              NrPredictionHeights = 1;
 double           PredictionHeights[1] = { 1.5 };

 // ---------------------- Initialisation of parameters ---------------- //
 WinProp_Structure_Init_ParameterMain(&GeneralParameters);

 for (int cell = 0; cell < MAX_CELLS; cell++)
 {
  WinProp_Structure_Init_Pattern(&WinPropPattern[cell]);
  WinProp_Structure_Init_Antenna(&Antenna[cell]);
  WinProp_Structure_Init_Carrier(&Carrier[cell]);
  WinProp_Structure_Init_Result(&PropResults[cell]);
 }
```

```
// ---------------- Definition of parameters -------------------------- //
// Definition of scenario.
const char* my_odb_database = API_DATA_FOLDER "outdoor/Helsinki"; // name of .oib
database without extensions
GeneralParameters.ScenarioMode = SCENARIOMODE_URBAN; // Urban prediction
GeneralParameters.PredictionModelUrban = PREDMODEL_IRT; // Use IRT Model
GeneralParameters.BuildingsMaterialsInd = 1;            // Individual material
properties for buildings
GeneralParameters.VegetationMode = VEGETATION_BUILDING; // vegetation from building
database
GeneralParameters.VegetationPropertiesInd = 1;          // Individual material
properties for vegetation

// Definition of prediction area.
GeneralParameters.UrbanLowerLeftX = 385828.35;
GeneralParameters.UrbanLowerLeftY = 6672097.054;
GeneralParameters.UrbanUpperRightX = 386012.35;
GeneralParameters.UrbanUpperRightY = 6672356.054;

// Copy coordinates to prediction area of second model (not yet used).
GeneralParameters.RuralLowerLeftX = GeneralParameters.UrbanLowerLeftX;
GeneralParameters.RuralLowerLeftY = GeneralParameters.UrbanLowerLeftY;
GeneralParameters.RuralUpperRightX = GeneralParameters.UrbanUpperRightX;
GeneralParameters.RuralUpperRightY = GeneralParameters.UrbanUpperRightY;

// Size of matrix with results.
GeneralParameters.Resolution = 1.0;                         // Resolution in meter
GeneralParameters.NrLayers = NrPredictionHeights;           // Number of prediction
heights
GeneralParameters.PredictionHeights = PredictionHeights;    // Prediction height in
meter

// Building vector data and topography.
GeneralParameters.BuildingsMode = BUILDINGSMODE_IRT; // load a .oib database
sprintf(GeneralParameters.BuildingsName, "%s", my_odb_database);

// Break point parameters
GeneralParameters.BreakpointMode = BREAKPOINT_DEFAULT;
GeneralParameters.BreakpointFactor = 4.;
GeneralParameters.BreakpointOffset = 0;

// Selection of paths
GeneralParameters.MaxPathLossEnabled = 1;
GeneralParameters.MaxPathLoss = 200.f;
GeneralParameters.MaxDynamicPathsEnabled = 1;
GeneralParameters.MaxDynamicPaths = 100.f;
GeneralParameters.MaxNumberPathsUsed = 1;
GeneralParameters.MaxNumberPaths = 20;

// Antenna patten
WinPropPattern[0].Mode = PATTERN_MODE_FILE;   // Load pattern from file
sprintf(WinPropPattern[0].Filename, "%s",
API_DATA_FOLDER "antennas/3750MHz_18dBi.msi"); // Pattern file

// Desired propagation results in WinProp result format
WinProp_Propagation_Results OutputResults;
WinProp_Structure_Init_Propagation_Results(&OutputResults);
sprintf(OutputResults.ResultPath, "%s", API_DATA_FOLDER "output/
network_planning_5g_tdd/propagation"); // Output data directory
OutputResults.Delay = 1;
OutputResults.StatusLOS = 1;
OutputResults.AdditionalResultsASCII = 1;
GeneralParameters.OutputResults = &OutputResults;
```

```
// Callback functions.
   WinProp_Callback Callback;
   WinProp_Structure_Init_Callback(&Callback);
Callback.Percentage = CallbackProgress;
Callback.Message = CallbackMessage;
Callback.Error = CallbackError;

//-------------------------------------------------------------------------
// Compute wave propagation for cells
//-------------------------------------------------------------------------

// ----------------  Define IRT parameters -------------------------- //
Model_UrbanIRT ParameterIRT;
WinProp_Structure_Init_Model_UrbanIRT(&ParameterIRT);

// set path loss exponents
ParameterIRT.BreakpointExponentBeforeLOS = 2.3;
ParameterIRT.BreakpointExponentAfterLOS = 3.3;
ParameterIRT.BreakpointExponentBefore = 2.5;
ParameterIRT.BreakpointExponentAfter = 3.3;

// Number of interactions
ParameterIRT.MaxReflections = 2;
ParameterIRT.MaxDiffractions = 1;
ParameterIRT.MaxScatterings = 0;
ParameterIRT.MaxSumReflDiff = 2;

// Superposition and diffraction mode
ParameterIRT.Superposition = 0;
ParameterIRT.DiffractionModel = 'e';
ParameterIRT.PostProcessingMode = 0;
ParameterIRT.Postprocessing = 0;
ParameterIRT.FreeSpaceCancel = 0;

// -------------------------  antennas  ------------------------------- //
// Position and configuration of the antennas (cells).
char AntennaName[MAX_CELLS][500] = { "Cell_1"};
double SiteX[MAX_CELLS] = { 385925.79 };
double SiteY[MAX_CELLS] = { 6672148.00 };
double SiteZ[MAX_CELLS] = { 15. };
double Power[MAX_CELLS] = { 15. }; // Power in dBm
double Azmith[MAX_CELLS] = { 0. };
double Downtilt[MAX_CELLS] = { 10. };
double Frequency[MAX_CELLS] = { 3750.0 }; // same frequencies in MHz as will defined
for the carriers (for network planning)
int AntennaCarrier[MAX_CELLS] = { 1 };

for (int cell = 0; cell < MAX_CELLS; cell++)
{
 // set antenna pattern
 Antenna[cell].Pattern = &WinPropPattern[cell];

 // Set antenna properties now.
 AntennaPropertiesSet(&Antenna[cell], SiteX[cell], SiteY[cell], SiteZ[cell],
Frequency[cell], AntennaName[cell],
  Power[cell], WINPROP_POWER_OUTPUT, WINPROP_MODEL_IRT, Azmith[cell],
Downtilt[cell], cell + 1);

 Carrier[cell].CarrierID = AntennaCarrier[cell];
 Carrier[cell].NrAntennaBeams = 1;
 Carrier[cell].SymbolsPerSlot = 14;
```

```
  Carrier[cell].Numerology = 1;
  Carrier[cell].TDD_Slots_DL = 7;
  Carrier[cell].TDD_Slots_UL = 7;
  Carrier[cell].TDD_Slots_Flex = 0;
  Carrier[cell].BeamGainCtrlServer = 0.;
  Carrier[cell].BeamGainCtrlInterferer = 0.;
  Carrier[cell].BeamGainDataServer = 0.;
  Carrier[cell].BeamGainDataInterferer = 0.;

  CarrierPropertiesSet(&Antenna[cell], &Carrier[cell]);

  /*---------------------- Compute outdoor prediction ------------------------*/
  Error = OutdoorPlugIn_ComputePrediction(&Antenna[cell], &GeneralParameters, NULL,
 0, &ParameterIRT, NULL, NULL, NULL, &Callback, &PropResults[cell], NULL, NULL,
 NULL);

  /*---------------------- write propagation results ------------------------*/
  if (Error == 0)
  {
   char NameForOutput[200];
   sprintf(NameForOutput, "%s%s%s%s", API_DATA_FOLDER, "output/
network_planning_5g_tdd/propagation/", Antenna[cell].Name, ".txt");

   write_ascii(&PropResults[cell], NameForOutput);
  }
  else
  {
   /* Error during prediction. Print error message. */
   CallbackError(GeneralParameters.ErrorMessageMain, Error);
  }
 }


 // -------------------- Network Planning Project ---------------------- /

 // Create a new network planning project based on 5G TDD air interface
 if (Error == 0)
 {
  Error = WinProp_Net_Project_Open(&ProjectHandle, NET_AIRINTERFACE_5G_TDD_GENERIC,
 NULL);
 }

 // Add all propagation maps which have been computed.
 if (Error == 0)
 {
  int MapIndex[MAX_CELLS];
  for (int Count = 0; Count < MAX_CELLS; Count++)
  {
   if (Error == 0)
   {
    Error = WinProp_Net_PropagationMap_Add(ProjectHandle, &MapIndex[Count],
 &Antenna[Count], &PropResults[Count]);
   }
  }
 }

 // Set a name for the project.
 if (Error == 0)
 {
  char ProjectName[200];
  sprintf(ProjectName, "%s", "TDD-5G");
```

```
 Error = WinProp_Net_Project_Para_Set(ProjectHandle, NET_PARA_PROJECTNAME, NULL,
NULL, ProjectName);
}

// -------------------  numerology  ----------------------- //
// Set numerology and bandwidth
if (Error == 0)
{
 ParaValueInt = NET_PARA_5G_NUMEROLOGY1_100MHZ_FR1;
 Error = WinProp_Net_Project_Para_Set(ProjectHandle, NET_PARA_BANDWIDTH_MODE_5G,
NULL, &ParaValueInt, NULL);
}

// -------------------  TDD  ----------------------- //
if (Error == 0)
{
 ParaValueInt = 1; // TDD mode
 Error = WinProp_Net_Project_Para_Set(ProjectHandle, NET_PARA_DUPLEX_SUBMODE, NULL,
&ParaValueInt, NULL);

}

if (Error == 0)
{
 ParaValueInt = 3; // TDD properties specified for each carrier individually
 Error = WinProp_Net_Project_Para_Set(ProjectHandle, NET_PARA_TDD_VARIABLE_SWITCH,
NULL, &ParaValueInt, NULL);
}

// --------------------------  Carriers  ------------------------------- //
// Add carriers in n1 band.
if (Error == 0)
{
 double FrequencyUL = 0.;
 double FrequencyDL = 0.;
 double CarrierSeparation = 100.;
 for (int CurrentCarrier = 0; CurrentCarrier < 1; CurrentCarrier++)
 {
  FrequencyUL = 3750.0 + (CurrentCarrier * CarrierSeparation);
  FrequencyDL = 3750.0 + (CurrentCarrier * CarrierSeparation);
  Error = WinProp_Net_Carrier_Add(ProjectHandle, CurrentCarrier + 1);

  Error = WinProp_Net_Carrier_Para_Set(
   ProjectHandle, CurrentCarrier + 1, NET_PARA_CARRIER_FREQ_DL, &FrequencyDL, NULL,
NULL);
  Error = WinProp_Net_Carrier_Para_Set(
   ProjectHandle, CurrentCarrier + 1, NET_PARA_CARRIER_FREQ_UL, &FrequencyUL, NULL,
NULL);
 }
}

// Check if number of carriers is correct.
if (Error == 0)
{
 int NrCarriers = 0;
 Error = WinProp_Net_Project_Para_Get(ProjectHandle, NET_PARA_CARRIERS, NULL,
&NrCarriers, NULL);
 if (Error == 0)
 {
  if (NrCarriers != 1)
   Error = 1;
 }
```

```
}

// ------------------------  Transmission modes  --------------------------- //
// Define 5G transmission modes
if (Error == 0)
{
 const char* TransmissionMode_Name[SERVICES_5G] = {
           "00 - QPSK - R 120",
           "01 - QPSK - R 193",
           "02 - QPSK - R 308",
           "03 - QPSK - R 449",
           "04 - QPSK - R 602",
           "06 - 16 QAM - R 378",
           "07 - 16 QAM - R 490",
           "08 - 16 QAM - R 658",
           "09 - 64 QAM - R 466",
           "10 - 64 QAM - R 666",
           "11 - 64 QAM - R 873",
           "12 - 256 QAM - R 797",
           "13 - 256 QAM - R 948" };
 double              TransmissionMode_Bitrate[SERVICES_5G] = { 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0 };
 double              TransmissionMode_SNIR[SERVICES_5G] = { -3.0, -2.0, -1.0, 0.0,
2.0, 7.0, 10.0, 12.0, 13.0, 16.0, 18.0, 22.0,  27.0 };
 int                 TransmissionMode_Coderate_K[SERVICES_5G] = { 120, 193, 308,
449, 602, 378, 490, 658, 466, 666, 873, 797, 948 };
 int                 TransmissionMode_Coderate_N[SERVICES_5G] = { 1024, 1024, 1024,
1024, 1024, 1024, 1024, 1024, 1024, 1024, 1024, 1024, 1024 };
 int                 TransmissionMode_MCS[SERVICES_5G] = { 2, 2, 2, 2, 2, 4, 4, 4,
6, 6, 6, 8, 8 };
 double              TransmissionMode_Backoff[SERVICES_5G] = { 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 3.0, 3.0, 3.0, 3.0, 3.0 };
 int                 TransmissionMode_Resource_Blocks_DL[SERVICES_5G] = { 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1 };
 int                 TransmissionMode_Resource_Blocks_UL[SERVICES_5G] = { 1, 1, 1,
1, 1, 1, 1, 1, 1, 1 ,1, 1, 1 };
 double              TransmissionMode_Overhead_Ratio_DL[SERVICES_5G] = { 14., 14.,
14., 14., 14., 14., 14., 14., 14., 14., 14. , 14. };
 double              TransmissionMode_Overhead_Ratio_UL[SERVICES_5G] = { 8., 8., 8.,
8., 8., 8., 8., 8., 8., 8., 8., 8., 8. };

 for (int CurrentService = 0; CurrentService < SERVICES_5G; CurrentService++)
 {
  // Add new service.
  char ServiceName[100];
  sprintf(ServiceName, "%s", TransmissionMode_Name[CurrentService]);
  Error = WinProp_Net_TransmissionMode_Add(ProjectHandle, ServiceName,
CurrentService);

  // Set position/priority.
  ParaValueInt = SERVICES_5G - CurrentService;
  Error = WinProp_Net_TransmissionMode_Para_Set(
   ProjectHandle, CurrentService, NET_PARA_TRANS_MODE_POSITION, NULL, &ParaValueInt,
NULL);

  // Set bitrate.
  ParaValueDouble = TransmissionMode_Bitrate[CurrentService];
  Error = WinProp_Net_TransmissionMode_Para_Set(
   ProjectHandle, CurrentService, NET_PARA_TRANS_MODE_BITRATE_DL, &ParaValueDouble,
NULL, NULL);
  Error = WinProp_Net_TransmissionMode_Para_Set(
```

```
    ProjectHandle, CurrentService, NET_PARA_TRANS_MODE_BITRATE_UL, &ParaValueDouble,
NULL, NULL);

  // Set code rate.
  ParaValueInt = TransmissionMode_Coderate_K[CurrentService];
  Error = WinProp_Net_TransmissionMode_Para_Set(
   ProjectHandle, CurrentService, NET_PARA_TRANS_MODE_CODERATE_K_DL, NULL,
&ParaValueInt, NULL);
  Error = WinProp_Net_TransmissionMode_Para_Set(
   ProjectHandle, CurrentService, NET_PARA_TRANS_MODE_CODERATE_K_UL, NULL,
&ParaValueInt, NULL);

  ParaValueInt = TransmissionMode_Coderate_N[CurrentService];
  Error = WinProp_Net_TransmissionMode_Para_Set(
   ProjectHandle, CurrentService, NET_PARA_TRANS_MODE_CODERATE_N_DL, NULL,
&ParaValueInt, NULL);
  Error = WinProp_Net_TransmissionMode_Para_Set(
   ProjectHandle, CurrentService, NET_PARA_TRANS_MODE_CODERATE_N_UL, NULL,
&ParaValueInt, NULL);

  // Set number of resource blocks.
  ParaValueInt = TransmissionMode_Resource_Blocks_DL[CurrentService];
  Error = WinProp_Net_TransmissionMode_Para_Set(
   ProjectHandle, CurrentService, NET_PARA_TRANS_MODE_RESOURCE_BLOCKS_DL, NULL,
&ParaValueInt, NULL);
  ParaValueInt = TransmissionMode_Resource_Blocks_UL[CurrentService];
  Error = WinProp_Net_TransmissionMode_Para_Set(
   ProjectHandle, CurrentService, NET_PARA_TRANS_MODE_RESOURCE_BLOCKS_UL, NULL,
&ParaValueInt, NULL);

  // Set overhead ratio.
  ParaValueDouble = TransmissionMode_Overhead_Ratio_UL[CurrentService];
  Error = WinProp_Net_TransmissionMode_Para_Set(
   ProjectHandle, CurrentService, NET_PARA_TRANS_MODE_OVERHEAD_RATIO_UL,
&ParaValueDouble, NULL, NULL);
  ParaValueDouble = TransmissionMode_Overhead_Ratio_DL[CurrentService];
  Error = WinProp_Net_TransmissionMode_Para_Set(
   ProjectHandle, CurrentService, NET_PARA_TRANS_MODE_OVERHEAD_RATIO_DL,
&ParaValueDouble, NULL, NULL);

  // Set required SNIR.
  ParaValueDouble = TransmissionMode_SNIR[CurrentService];
  Error = WinProp_Net_TransmissionMode_Para_Set(
   ProjectHandle, CurrentService, NET_PARA_TRANS_MODE_SNIR_DL, &ParaValueDouble,
NULL, NULL);
  Error = WinProp_Net_TransmissionMode_Para_Set(
   ProjectHandle, CurrentService, NET_PARA_TRANS_MODE_SNIR_UL, &ParaValueDouble,
NULL, NULL);

  // Set modulation scheme.
  ParaValueInt = TransmissionMode_MCS[CurrentService];
  Error = WinProp_Net_TransmissionMode_Para_Set(
   ProjectHandle, CurrentService, NET_PARA_TRANS_MODE_MODULATION_UL, NULL,
&ParaValueInt, NULL);
  Error = WinProp_Net_TransmissionMode_Para_Set(
   ProjectHandle, CurrentService, NET_PARA_TRANS_MODE_MODULATION_DL, NULL,
&ParaValueInt, NULL);

  // Set power backoff.
  ParaValueDouble = TransmissionMode_Backoff[CurrentService];
  Error = WinProp_Net_TransmissionMode_Para_Set(
   ProjectHandle, CurrentService, NET_PARA_TRANS_MODE_POWER_BACKOFF_UL,
&ParaValueDouble, NULL, NULL);
```

```
   Error = WinProp_Net_TransmissionMode_Para_Set(
    ProjectHandle, CurrentService, NET_PARA_TRANS_MODE_POWER_BACKOFF_DL,
&ParaValueDouble, NULL, NULL);

   // Get bitrate in kBit/s for current transmission mode. Bitrate is automatically
computed based
   // on parameters previously set and general air interface parameters.
   Error = WinProp_Net_TransmissionMode_Para_Get(
    ProjectHandle, CurrentService, NET_PARA_TRANS_MODE_BITRATE_DL, &ParaValueDouble,
NULL, NULL);
   TransmissionMode_Bitrate[CurrentService] = ParaValueDouble;
  }
 }

 // Check if number of services is correct.
 if (Error == 0)
 {
  int NrServices = 0;
  Error = WinProp_Net_Project_Para_Get(ProjectHandle, NET_PARA_SERVICES, NULL,
&NrServices, NULL);
  if (Error == 0)
  {
   if (NrServices != SERVICES_5G)
    Error = 1;
  }
 }

 // --------------------------  Project parameters  -------------------------- //

 // Set resolution for result matrix.
 if (Error == 0)
 {
  ParaValueDouble = GeneralParameters.Resolution;
  Error = WinProp_Net_Project_Para_Set(ProjectHandle, NET_PARA_RESOLUTION,
&ParaValueDouble, NULL, NULL);
 }
 // Set the simulation area
 if (Error == 0)
 {
  ParaValueDouble = GeneralParameters.UrbanLowerLeftX;
  Error = WinProp_Net_Project_Para_Set(ProjectHandle, NET_PARA_AREA_LL_X,
&ParaValueDouble, NULL, NULL);
 }
 if (Error == 0)
 {
  ParaValueDouble = GeneralParameters.UrbanLowerLeftY;
  Error = WinProp_Net_Project_Para_Set(ProjectHandle, NET_PARA_AREA_LL_Y,
&ParaValueDouble, NULL, NULL);
 }
 if (Error == 0)
 {
  ParaValueDouble = GeneralParameters.UrbanUpperRightX;
  Error = WinProp_Net_Project_Para_Set(ProjectHandle, NET_PARA_AREA_UR_X,
&ParaValueDouble, NULL, NULL);
 }
 if (Error == 0)
 {
  ParaValueDouble = GeneralParameters.UrbanUpperRightY;
  Error = WinProp_Net_Project_Para_Set(ProjectHandle, NET_PARA_AREA_UR_Y,
&ParaValueDouble, NULL, NULL);
 }
```

```
// Set paths for additional output in WinProp file format.
if (Error == 0)
{
 Error = WinProp_Net_Project_Para_Set(ProjectHandle, NET_PARA_OUTPUT_WINPROP, NULL,
NULL, API_DATA_FOLDER "output/network_planning_5g_tdd/network");
}

// ------------------------  Cell assignment   ------------------------------ //
if (Error == 0)
{
 int IntValue = 0; // highest Rx power for all carries mode
 Error = WinProp_Net_Project_Para_Set(ProjectHandle, NET_PARA_CELL_ASSIGNMENT_MODE,
NULL, &IntValue, NULL);
}

// Set Min required SNIR
if (Error == 0)
{
 ParaValueDouble = -5.4;
 Error = WinProp_Net_Project_Para_Set(ProjectHandle,
NET_PARA_CELL_ASSIGNMENT_MIN_REQ_SNIR, &ParaValueDouble, NULL, NULL);
}

// Set Min required power
if (Error == 0)
{
 ParaValueDouble = -95.;
 Error = WinProp_Net_Project_Para_Set(ProjectHandle,
NET_PARA_CELL_ASSIGNMENT_MIN_REQ_POWER, &ParaValueDouble, NULL, NULL);
}


// ------------------------  OFDM parameters  ------------------------------- //
// Set control sub-carriers
if (Error == 0)
{
 ParaValueInt = 2500;
 Error = WinProp_Net_Project_Para_Set(ProjectHandle, NET_PARA_SUBCARRIERS_CONTROL,
NULL, &ParaValueInt, NULL);
}

// Set reference sub-carriers
if (Error == 0)
{
 ParaValueInt = 500;
 Error = WinProp_Net_Project_Para_Set(ProjectHandle, NET_PARA_SUBCARRIERS_REFERENCE,
NULL, &ParaValueInt, NULL);
}

// Set Guard sub-carriers
if (Error == 0)
{
 ParaValueInt = 29;
 Error = WinProp_Net_Project_Para_Set(ProjectHandle, NET_PARA_SUBCARRIERS_GUARD,
NULL, &ParaValueInt, NULL);
}

// Set data symbols
if (Error == 0)
{
 ParaValueInt = 10;
```

```
 Error = WinProp_Net_Project_Para_Set(ProjectHandle, NET_PARA_SYMBOLS_DATA, NULL,
&ParaValueInt, NULL);
}

// Set control symbols
if (Error == 0)
{
 ParaValueInt = 1;
 Error = WinProp_Net_Project_Para_Set(ProjectHandle, NET_PARA_SYMBOLS_CONTROL, NULL,
&ParaValueInt, NULL);
}

// Set reference symbols
if (Error == 0)
{
 ParaValueInt = 3;
 Error = WinProp_Net_Project_Para_Set(ProjectHandle, NET_PARA_SYMBOLS_REFERENCE,
NULL, &ParaValueInt, NULL);
}

// Set FFT order
if (Error == 0)
{
 ParaValueInt = 4096;
 Error = WinProp_Net_Project_Para_Set(ProjectHandle, NET_PARA_FFT_SIZE, NULL,
&ParaValueInt, NULL);
}

// Desired network planning results in WinProp result format
if (Error == 0)
{
 Error = WinProp_Net_Project_Result_Switch(ProjectHandle, NET_RESULT_BEST_SERVER,
1);
 Error = WinProp_Net_Project_Result_Switch(ProjectHandle, NET_RESULT_MAX_DATARATE,
1);
 Error = WinProp_Net_Project_Result_Switch(ProjectHandle, NET_RESULT_MAX_THROUGHPUT,
1);
 Error = WinProp_Net_Project_Result_Switch(ProjectHandle, NET_RESULT_SNIR, 1);
}

//-----------------------------------------------------------------------------
//  Compute network planning results
//-----------------------------------------------------------------------------
if (Error == 0)
{
 char HeightString[200];
 sprintf(HeightString, "%s", "");

 // Generate string with height values, e.g. a string like "1.5 2.5 3.5".
 for (int Height = 0; Height < NrPredictionHeights; Height++)
 {
  // Add current height to string.
  char thisHeightStr[50];
  sprintf(thisHeightStr, "%.2f ", PredictionHeights[Height]);
  strcat(HeightString, thisHeightStr);
 }

 // Send heights to WinProp API.
 Error = WinProp_Net_Project_Para_Set(ProjectHandle, NET_PARA_HEIGHT_MULTIPLE, NULL,
NULL, HeightString);

 // Start network computation.
```

```
  if (Error == 0)
  {
   Error = WinProp_Net_Project_Compute(ProjectHandle, &Callback);
  }
 }

 // ------------------------------------------------------------------------
 // Retrieve  results
 // ------------------------------------------------------------------------
 // As an example: retrieve max. throughput (kbps) per pixel.
 if (Error == 0)
 {
  WinProp_Result* MaxThroughput = NULL;
  Error = WinProp_Net_NetworkMap_Get(ProjectHandle, -1, NET_RESULT_MAX_THROUGHPUT,
 &MaxThroughput);

  // Write max. throughput result to ASCII file.
  if (Error == 0)
  {
   char NameForOutput[200];
   sprintf(NameForOutput, "%s%s", API_DATA_FOLDER, "output/network_planning_5g_tdd/
network/Max Throughput DownLink.txt");
   write_ascii(MaxThroughput, NameForOutput);
  }

 }

 // As another example: retrieve SNIR (dB) per pixel.
 if (Error == 0)
 {
  WinProp_Result* MaxSNIR = NULL;
  Error = WinProp_Net_NetworkMap_Get(ProjectHandle, -1, NET_RESULT_SNIR, &MaxSNIR);

  // Write max. throughput result to ASCII file.
  if (Error == 0)
  {
   char NameForOutput[200];
   sprintf(NameForOutput, "%s%s", API_DATA_FOLDER, "output/network_planning_5g_tdd/
network/Max SNIR.txt");
   write_ascii(MaxSNIR, NameForOutput);
  }
 }

 // ------------------------------------------------------------------------
 // Free memory
 // ------------------------------------------------------------------------
 for (int Count = 0; Count < MAX_CELLS; Count++)
 {
  // Free propagation results.
  WinProp_FreeResult(&PropResults[Count]);
 }

 // Close network project.
 Error = WinProp_Net_Project_Close(ProjectHandle);

 return 0;
}

int _STD_CALL CallbackMessage(const char* Text)
{
 if (Text == nullptr)
  return 0;
```

```
 std::cout << "\n" << Text;

 return(0);
}

int _STD_CALL CallbackError(const char* Text, int Error)
{
 if (Text == nullptr)
  return 0;

 std::cout << "\n";

#ifdef __LINUX
 std::cout << "\033[31m" << "Error (" << Error << "): "; // highlight error in red
 color
#else
 HANDLE hConsole = GetStdHandle(STD_OUTPUT_HANDLE);
 SetConsoleTextAttribute(hConsole, FOREGROUND_RED);
 std::cout << "Error (" << Error << "): ";
#endif // __LINUX
 std::cout << Text;

#ifdef __LINUX
 std::cout << "\033[0m"; // highlight error in red color
#else
 SetConsoleTextAttribute(hConsole, FOREGROUND_RED | FOREGROUND_BLUE |
 FOREGROUND_GREEN);
#endif // __LINUX

 return 0;
}

int _STD_CALL CallbackProgress(int value, const char* text)
{
 char Line[200];

 sprintf(Line, "\n%d%% %s", value, text);
 std::cout << Line;

 return(0);
}

// Helper functions
void write_ascii(const WinProp_Result* Resultmatrix, const char* Filename) {

 if (Resultmatrix)
 {
  FILE* OutputFile = fopen(Filename, "w");
  if (OutputFile)
  {
   /* Loop through WinPropall pixels. */
   for (int x = 0; x < Resultmatrix->Columns; x++)
   {
    for (int y = 0; y < Resultmatrix->Lines; y++)
    {
     /* Compute real coordinates. */
     double Coordinate_X = Resultmatrix->LowerLeftX + ((double)x + 0.5) *
 Resultmatrix->Resolution;
     double Coordinate_Y = Resultmatrix->LowerLeftY + ((double)y + 0.5) *
 Resultmatrix->Resolution;
```

```
      /* Check if pixel was computed or not */
      if (Resultmatrix->Matrix[0][x][y] > -1000)
      {
       fprintf(OutputFile, "%.5f\t%.5f\t%.4f\n", Coordinate_X, Coordinate_Y,
 Resultmatrix->Matrix[0][x][y]);
      }
      else
      {
       fprintf(OutputFile, "%.5f\t%.5f\t%s\n", Coordinate_X, Coordinate_Y, "N.C.");
      }
     }
    }

    /* Close file. */
    fclose(OutputFile);
   }
   else
   {
    printf("\nCould not open the File: %s for writing.\n", Filename);
   }
  }
}

void CarrierPropertiesSet(WinProp_Antenna* Antenna, const WinProp_Carrier* Carrier) {

 Antenna->Carriers.CarrierID = Carrier->CarrierID;
 Antenna->Carriers.SystemID = Carrier->SystemID;
 Antenna->Carriers.CellLoad = Carrier->CellLoad;
 Antenna->Carriers.MimoID = Carrier->MimoID;

 Antenna->Carriers.NoiseRiseUL = Carrier->NoiseRiseUL;
 Antenna->Carriers.PowerBackoffPilotDL = Carrier->PowerBackoffPilotDL;

 // 5G Carrier parameters
 Antenna->Carriers.Numerology = Carrier->Numerology;
 Antenna->Carriers.SymbolsPerSlot = Carrier->SymbolsPerSlot;
 Antenna->Carriers.NrAntennaBeams = Carrier->NrAntennaBeams;
 Antenna->Carriers.BeamGainCtrlServer = Carrier->BeamGainCtrlServer;
 Antenna->Carriers.BeamGainCtrlInterferer = Carrier->BeamGainCtrlInterferer;
 Antenna->Carriers.BeamGainDataServer = Carrier->BeamGainDataServer;
 Antenna->Carriers.BeamGainDataInterferer = Carrier->BeamGainDataInterferer;
 Antenna->Carriers.TDD_Slots_DL = Carrier->TDD_Slots_DL;
 Antenna->Carriers.TDD_Slots_UL = Carrier->TDD_Slots_UL;
 Antenna->Carriers.TDD_Slots_Flex = Carrier->TDD_Slots_Flex;
}
```

2.6.15 Propagation in Rural Scenarios

Detailed Description

This is an example of how to use the WinProp API for predictions in a rural scenario. The full example is distributed with the installation.

```
#include <stdio.h>
#include <string>
```

```
#include <iostream>

#include "rural_propagation.h"

#ifndef API_DATA_FOLDER
#define API_DATA_FOLDER "../../api/winprop/data/"
#endif // !API_DATA_FOLDER

int main(int argc, char** argv)
{
    int                          Error = 0;
    WinProp_ParaMain             GeneralParameters;
    WinProp_ParaRural            RuralParameters;
    WinProp_Antenna              Antenna;
    WinProp_Callback             Callback;
    WinProp_Result               Resultmatrix;
    WinProp_Propagation_Results OutputResults;

    /*****************************************************************************/
    /* Initialization of parameters                                              */
    /*****************************************************************************/

    WinProp_Structure_Init_ParameterMain(&GeneralParameters);
    WinProp_Structure_Init_ParameterRural(&RuralParameters);
    WinProp_Structure_Init_Antenna(&Antenna);
    WinProp_Structure_Init_Callback(&Callback);
    WinProp_Structure_Init_Result(&Resultmatrix);
    WinProp_Structure_Init_Propagation_Results(&OutputResults);

    /*****************************************************************************/
    /* Load clutter data                                                         */
    /*****************************************************************************/

    Error = InterfaceLoadClutterASC(
        &GeneralParameters.Clutter,
        API_DATA_FOLDER "outdoor/RuralClutter.asc",
        API_DATA_FOLDER "outdoor/RuralClutter.table");

    /*****************************************************************************/
    /* Load topo data                                                            */
    /*****************************************************************************/

    Error = InterfaceLoadTopoASC(
        &GeneralParameters.Topography, API_DATA_FOLDER "outdoor/RuralTopo.asc");

    /*****************************************************************************/
    /* Definition of scenario (topo database in WinProp format)                  */
    /*****************************************************************************/

    /* Definition of general parameters. */
    GeneralParameters.ScenarioMode = SCENARIOMODE_RURAL;
    GeneralParameters.PredictionModelRural = PREDMODEL_RURAL_2RAY_DET;

    /* Definition of prediction area. */
    GeneralParameters.UrbanLowerLeftX = GeneralParameters.Topography.lowerLeftX;
    GeneralParameters.UrbanLowerLeftY = GeneralParameters.Topography.lowerLeftY;
    GeneralParameters.UrbanUpperRightX = GeneralParameters.Topography.upperRightX;
    GeneralParameters.UrbanUpperRightY = GeneralParameters.Topography.upperRightY;

    /* Copy coordinates to prediction area of second model (not yet used) */
    GeneralParameters.RuralLowerLeftX = GeneralParameters.UrbanLowerLeftX;
    GeneralParameters.RuralLowerLeftY = GeneralParameters.UrbanLowerLeftY;
```

```
    GeneralParameters.RuralUpperRightX = GeneralParameters.UrbanUpperRightX;
    GeneralParameters.RuralUpperRightY = GeneralParameters.UrbanUpperRightY;

    double PredictionHeight = 1.5;

    /* Size of matrix with results. */
    GeneralParameters.Resolution = 10.0;                        // Resolution in meter
    GeneralParameters.NrLayers = 1;                             // Number of
 prediction heights
    GeneralParameters.PredictionHeights = &PredictionHeight;    // Prediction height
 in meter

    /* No vector buildings but clutter and topo. */
    GeneralParameters.BuildingsMode = BUILDINGSMODE_NONE;
    GeneralParameters.TopographyMode = TOPOMODE_RAM;
    GeneralParameters.ClutterMode = CLUTTERMODE_RAM;
    GeneralParameters.ClutterConsideration = CLUTTERLOSS_HEIGHTS_NONE;

    /* Definition of outputs to be computed and written in WinProp format. */
    GeneralParameters.OutputResults = &OutputResults;
    sprintf(OutputResults.ResultPath, "%s", API_DATA_FOLDER "output/
rural_propagation"); // Output data directory
 OutputResults.FieldStrength = 1;

    /*************************************************************************/
    /* Definition of antenna.                                                */
    /*************************************************************************/

    /* Position and configuration. */
    Antenna.Id = 1;
    Antenna.SiteId = 1;
    Antenna.Longitude_X = 321900.0;
    Antenna.Latitude_Y = 4158000.0;
    Antenna.Height = 40.0;
    Antenna.Power = 43.0; // Power in dBm
    Antenna.Frequency = 1800.0; // Frequency in MHz
    sprintf(Antenna.Name, "%s", "Antenna 1");

    /*************************************************************************/
    /* Definition of callback functions (for progress messages)              */
    /*************************************************************************/

    /* Two functions: Message output and progress bar update. */
    Callback.Percentage = CallbackProgress;
    Callback.Message = CallbackMessage;
    Callback.Error = CallbackError;

    /*************************************************************************/
    /* Compute prediction now                                                */
    /*************************************************************************/

    /* Start computation within DLLs. */
    if (Error == 0)
        Error = OutdoorPlugIn_ComputePrediction(
            &Antenna, &GeneralParameters, NULL, 0, NULL, NULL, &RuralParameters,
 NULL, &Callback, &Resultmatrix, NULL, NULL, NULL);

    /*************************************************************************/
    /* Save prediction result to ASCII file.                                 */
    /*************************************************************************/

    if (Error == 0)
```

```
    {
        /* Write result matrix to ASCII file. */
        write_ascii(&Resultmatrix, API_DATA_FOLDER "output/rural_propagation/
Output_Rural.txt");
    }

    /*************************************************************************/
    /* Free matrix for results and buildings                                 */
    /*************************************************************************/

    WinProp_FreeResult(&Resultmatrix);
    InterfaceClutterFree(&GeneralParameters.Clutter);
    InterfaceTopoFree(&GeneralParameters.Topography);

    return Error == 0 ? EXIT_SUCCESS : EXIT_FAILURE;
}

int _STD_CALL CallbackMessage(const char * Text)
{
 if (Text == nullptr)
  return 0;

 std::cout << "\n" << Text;

 return(0);
}

int _STD_CALL CallbackError(const char * Text, int Error)
{
 if (Text == nullptr)
  return 0;

 std::cout << "\n";

#ifdef __LINUX
 std::cout << "\033[31m" << "Error (" << Error << "): "; // highlight error in red
 color
#else
 HANDLE hConsole = GetStdHandle(STD_OUTPUT_HANDLE);
 SetConsoleTextAttribute(hConsole, FOREGROUND_RED);
 std::cout << "Error (" << Error << "): ";
#endif // __LINUX
 std::cout << Text;

#ifdef __LINUX
 std::cout << "\033[0m"; // highlight error in red color
#else
 SetConsoleTextAttribute(hConsole, FOREGROUND_RED | FOREGROUND_BLUE |
 FOREGROUND_GREEN);
#endif // __LINUX

 return 0;
}

int _STD_CALL CallbackProgress(int value, const char* text)
{
 char Line[200];

 sprintf(Line, "\n%d%% %s", value, text);
 std::cout << Line;

 return(0);
```

```
}

void write_ascii(const WinProp_Result* Resultmatrix, const char* Filename) {
 FILE* OutputFile = fopen(Filename,"w");
 if (OutputFile)
 {
  /* Loop all pixels. */
  for (int x = 0; x < Resultmatrix->Columns; x++)
  {
   for (int y = 0; y < Resultmatrix->Lines; y++)
   {
    /* Compute real coordinates. */
    double Coordinate_X = Resultmatrix->LowerLeftX + ((double)x + 0.5) *
 Resultmatrix->Resolution;
    double Coordinate_Y = Resultmatrix->LowerLeftY + ((double)y + 0.5) *
 Resultmatrix->Resolution;

    /* Check if pixel was computed or not */
    if (Resultmatrix->Matrix[0][x][y] > -1000)
     fprintf(OutputFile, "%.2f\t%.2f\t%.2f\n", Coordinate_X, Coordinate_Y,
 Resultmatrix->Matrix[0][x][y]);
   }
  }

  /* Close file. */
  fclose(OutputFile);
 }
 else
  printf("\nCould not open the File: %s for writing.\n",Filename);
}
```

2.6.16 FMCW Radar Postprocessing in Time-variant Indoor Scenarios

Detailed Description

This is an example of how to use the WInProp API for FMCW radar postprocessing in a time-variant indoor scenario. The full example is distributed with the installation.

```
#include <cstdio>
#include <string>
#include <iostream>

#include "indoor_propagation_fmcw.h"

#include "../SuperposeMS/winprop_superposems.hpp"
#include "../SuperposeMS/winprop_superposems_engine.h"

#include "../Interface/Init.h"
#include "../FMCW/winprop_fmcw_radar.h"

#ifndef API_DATA_FOLDER
#define API_DATA_FOLDER "../../api/winprop/data/"
#endif // !API_DATA_FOLDER
```

```
int main(int argc, char** argv)
{
 int Error = 0;

 /* ------------------ Initialization of scenario --------------------------- */
 WinProp_Scenario WinPropScenario;
 WinProp_Structure_Init_Scenario(&WinPropScenario);
 /* ---------- Load indoor vector database and initialize scenario ------------ */
 /* Assign database name. */
 WinPropScenario.Scenario = WINPROP_SCENARIO_INDOOR;
 sprintf(WinPropScenario.VectorDatabase, "%s", API_DATA_FOLDER "../data/indoor/
RadarScenario.idb");

 /* Define callback functions. */
 WinProp_Callback WinPropCallback;
 WinProp_Structure_Init_Callback(&WinPropCallback);
 WinPropCallback.Percentage = CallbackProgress;
 WinPropCallback.Message = CallbackMessage;
 WinPropCallback.Error = CallbackError;

 /* ------------------------- Set up prediction ------------------------------ */
 if (Error == 0)
 {
  /* Definition of prediction points. */
  int NrPoints = 1;
  WinProp_Receiver WinPropReceiver;
  WinProp_Structure_Init_Receiver(&WinPropReceiver);
  WinPropReceiver.Location.x = 5.8;
  WinPropReceiver.Location.y = -2.;
  WinPropReceiver.Location.z = 0.5;
  WinPropReceiver.GroupID = 230; // Group id of the radar car

  /* Definition of antenna pattern. */
  WinProp_Pattern WinPropPattern;
  WinProp_Structure_Init_Pattern(&WinPropPattern);
  WinPropPattern.Mode = PATTERN_MODE_FILE; // Load pattern from file
  sprintf(WinPropPattern.Filename, "%s", API_DATA_FOLDER "antennas/
RadarAntenna.ffe");// Pattern file (including extension)

  /* Definition of antenna properties. */
  WinProp_Antenna WinPropAntenna;
  WinProp_Structure_Init_Antenna(&WinPropAntenna);
  WinPropAntenna.Enabled = ANTENNA_ENABLED;
  WinPropAntenna.Id = 1;
  WinPropAntenna.SiteId = 1;
  WinPropAntenna.Longitude_X = 5.8;
  WinPropAntenna.Latitude_Y = -2.0;
  WinPropAntenna.Height = 0.5; // Antenna height 0.5 meter
  WinPropAntenna.Model = WINPROP_MODEL_SRT; // Use the standard ray tracing
  WinPropAntenna.DataType = PROP_RESULT_POWER; // Compute received power
  WinPropAntenna.Power = 13.; // Power in dBm
  WinPropAntenna.PowerMode = WINPROP_POWER_OUTPUT; // Power mode
  WinPropAntenna.Frequency = 77000.; // Frequency 77 GHz
  WinPropAntenna.GroupIDMounted = 230; // Group id of the radar car

  sprintf(WinPropAntenna.Name, "%s", "radar_antenna"); // name of the antenna

  /* Definition of ray tracing parameters. */
  Model_RAYTRACING ModelRT;
  WinProp_Structure_Init_Model_RayTracing(&ModelRT);
  ModelRT.MaxNrDiffractions = 1;
```

```
  ModelRT.MaxNrReflections = 1;
  ModelRT.MaxNrTransmissions = 0;
  ModelRT.MaxNrReflDiffr = 1;
  ModelRT.Superposition = 1;
  ModelRT.ComputeAlwaysDirectRay = 2;

  /* Definition of outputs to be computed and written in WinProp format. */
  WinProp_Additional WinPropMore;
  WinProp_Propagation_Results OutputResults;
  WinProp_Structure_Init_Additional(&WinPropMore);
  WinProp_Structure_Init_Propagation_Results(&OutputResults);
  WinPropMore.OutputResults = &OutputResults;
  OutputResults.RayFilePropPaths = 1;
  OutputResults.StrFilePropPaths = 1;

  /* Further parameters: */
  WinPropMore.InteractionModel = WINPROP_INTERACTION_MODEL_GTDUTD;
  WinPropMore.MaxNumberPathsUsed = 1;
  WinPropMore.MaxNumberPaths = 64;
  WinPropMore.MaxDynamicPathsEnabled = 1;
  WinPropMore.MaxDynamicPaths = 100.;
  WinPropMore.MaxPathLossEnabled = 1;
  WinPropMore.MaxPathLoss = 200.;

  /* Set time instance(s) and output folder */
  double timeInstances[1] = { 1.0 }; // 1 s
  WinPropMore.TimeInstances = timeInstances;
  WinPropMore.NbrTimeInstances = 1;
        sprintf(OutputResults.ResultPath, "%s", API_DATA_FOLDER "output/
indoor_propagation_fmcw"); // name of the antenna

  /* Call the WinProp API to open a project and load the vector database. */
  int ProjectHandle = 0;
  Error = WinProp_Open(&ProjectHandle, &WinPropScenario, &WinPropCallback);

  /* ----- Start prediction (including loop over all time instances) ------ */
  WinProp_ResultPointsList* PowerResult = nullptr;
  Error = WinProp_Predict_Points(
   ProjectHandle,
   &WinPropAntenna,
   &WinPropReceiver,
   NrPoints,
   &ModelRT,
   &WinPropMore,
   &PowerResult,
   nullptr,
   nullptr);

  if (Error == 0)
  {
   char PowerOutputFile[200];
   sprintf(PowerOutputFile, API_DATA_FOLDER "output/indoor_propagation_fmcw/
%s_Power.txt", WinPropAntenna.Name);
   write_ascii(PowerResult, PowerOutputFile);
  }

  // Optionally, create range-angle heat maps
  bool generateRangeAngle = true;
        constexpr int nbrRxElements = 4; // nr of RX elements in a linear array,
 relevant for range-angle heat maps
        constexpr int nbrTxElements = 4; // nr of TX elements in a linear array,
 relevant for range-angle heat maps
```

```
        constexpr int nbrElements = nbrRxElements * nbrTxElements;
  float rxSpacing = 0.5f; // elements spacing in [lambda], relevant for range-angle
 heat maps
        float txSpacing = 2.f;  // elements spacing in [lambda], relevant for range-
angle heat maps

  WinProp_RayMatrixElement subStreamRays[nbrElements]; // ray matrices for sub-stream
 results, relevant for range-angle heat maps
  for (int i = 0; i < nbrElements; i++)
   WinProp_Structure_Init_RayMatrixElement(&subStreamRays[i]);

  if (generateRangeAngle)
  {
   /*----------------------- Rx antenna superposition (RunMS) --------------------*/

   // Create project
   WinProp_SuperposeMS SuperposeMS;

   // Setup tx array
   WINPROP_MS_ARRAY_TYPE txArrayAPI = WINPROP_MS_ARRAY_LINEAR;
   WINPROP_MS_POLARIZATION_TYPE pol = WINPROP_MS_POLARIZATION_VERTICAL;

   if (Error == 0)
    Error = SuperposeMS.setArray(
                    false,
                    txArrayAPI /* array Type*/,
                    nbrTxElements /* nrElements */,
                    270.f /* array azimuth */,
                    txSpacing /* tx spacing */,
                    0.f /* radius*/,
                    false /* considerAntennaCoupling */);

            if (Error == 0)
            {
                // set individual tx elements
                for (int tx = 0; tx < nbrTxElements && Error == 0; tx++)
                {
                    // element azimuth with east over north orientation (overall
 direction: east when adding array Azimuth),
                    // 90deg tilt is equal to horizontal orientation.
                    Error = SuperposeMS.setArrayElement(
                        false, tx, &WinPropPattern, nullptr,
                        90.f/*azimuth*/, 90.f/*tilt*/,
                        pol, { 0., 0., 0. }/*offset*/);
                }
            }

   // Setup rx array
   WINPROP_MS_ARRAY_TYPE rxArrayAPI = WINPROP_MS_ARRAY_LINEAR;
   if (Error == 0) {
    Error = SuperposeMS.setArray(true,
     rxArrayAPI, /* array Type*/
     nbrRxElements /* nrElements */,
     270.f /* array azimuth */,
     rxSpacing /*rx spacing */,
     0.f /* radius*/,
     false /* considerAntennaCoupling */);
   }

   // set individual rx elements
   for (int rx = 0; rx < nbrRxElements && Error == 0; rx++)
   {
```

```
    // element azimuth with east over north orientation (overall direction: east when
 adding array Azimuth),
    // 90deg tilt is equal to horizontal orientation.
    Error = SuperposeMS.setArrayElement(
     true, rx, &WinPropPattern, nullptr,
     90.f/*azimuth*/, 90.f/*tilt*/,
     pol, { 0., 0., 0. }/*offset*/);
   }

   // Set computation parameter
   WinProp_MS_Para msPara;
   WinProp_Structure_Init_MS_Para(&msPara);
   msPara.coherentSuperposition = 1;
   msPara.nrRays = 64;

   // Enable ouputs
   WinProp_MS_AdditionalResults msAdditionalResult;
   WinProp_Structure_Init_MS_AdditionalResults(&msAdditionalResult);
   msAdditionalResult.propagationPaths = 1;
   msAdditionalResult.perSubChannelPower = 1;
   sprintf(msAdditionalResult.outputPath, "%s%s", API_DATA_FOLDER, "output/
indoor_propagation_fmcw");

   // set propagation points
   if (Error == 0)
    Error = SuperposeMS.setPropagationPoints(0, WinPropAntenna, *PowerResult);

   // compute RunMS
   if (Error == 0)
    Error = SuperposeMS.compute(msPara, &msAdditionalResult, &WinPropCallback);

   // get sub-stream results
            int ele = 0;
   for (int tx = 0; tx < nbrTxElements && Error == 0; tx++)
   {
               for (int rx = 0; rx < nbrRxElements && Error == 0; rx++)
               {
                   WinProp_ResultPointsList subStreamRes;
                   WinProp_Structure_Init_ResultPointsList(&subStreamRes);
                   char subStreamResPower[1024];
                   char subStreamResPowerASCII[1024];
                   char subStreamResRays[1024];

                   if (nbrTxElements == 1)
                   {
                       sprintf(subStreamResPower, "%s/%s Sub-Channel Rx-%d
 Power.fpp", msAdditionalResult.outputPath, WinPropAntenna.Name, rx + 1);
                       sprintf(subStreamResPowerASCII, "%s/%s Sub-Channel Rx-%d
 Power.txt", msAdditionalResult.outputPath, WinPropAntenna.Name, rx + 1);
                       sprintf(subStreamResRays, "%s/%s Sub-Channel Rx-%d Rays.ray",
 msAdditionalResult.outputPath, WinPropAntenna.Name, rx + 1);
                   }
                   else if (nbrTxElements > 1)
                   {
                       sprintf(subStreamResPower, "%s/%s Tx-%d Sub-Channel Rx-%d
 Power.fpp", msAdditionalResult.outputPath, WinPropAntenna.Name, tx + 1, rx + 1);
                       sprintf(subStreamResPowerASCII, "%s/%s Tx-%d  Sub-Channel Rx-
%d Power.txt", msAdditionalResult.outputPath, WinPropAntenna.Name, tx + 1, rx + 1);
                       sprintf(subStreamResRays, "%s/%s Tx-%d Sub-Channel Rx-%d
 Rays.ray", msAdditionalResult.outputPath, WinPropAntenna.Name, tx + 1, rx + 1);
                   }
```

```
                    Error = WinProp_ResultPoints_Read(&subStreamRes,
 subStreamResPower, subStreamResRays);

                    if (Error == 0)
                    {
                        // Write to ascii file
                        write_ascii(&subStreamRes, subStreamResPowerASCII);
                        // move rays
                        subStreamRays[ele++] = subStreamRes.ResultPoints[0].Rays;
                        subStreamRes.ResultPoints[0].Rays.NrRays = 0;
                        subStreamRes.ResultPoints[0].Rays.Rays = nullptr;
                    }

                    WinProp_Structure_Free_ResultPointsList(&subStreamRes);
                }
   }
  }

  if (Error == 0)
  {
   /* ------------------------- FMCW radar ------------------------------ */

   /* Definition of radar parameters */
   WinProp_FMCW_Para RadarPara;
   RadarPara.Mode = WinProp_FMCW_Para::ParametersMode::SET_PARAMETERS; // set
 parameters mode
   RadarPara.Freq = 77e9; // radar frequency
   RadarPara.Tc = 20e-6; // chirp duration
   RadarPara.B = 1e9; // sweep bandwidth
   RadarPara.NrChirps = 64; // number of chirps
   RadarPara.Coherent = true; // coherent superposition of rays
   RadarPara.NrBins = WinProp_FMCW_Para::FreqBins::FFT_512; // FFT Samples (range
 domain)
   RadarPara.NrRx = generateRangeAngle ? nbrRxElements : 1;
            double lambda = 2.99792458e8 / (WinPropAntenna.Frequency * 1.e6);
   RadarPara.RxSpacing = rxSpacing * lambda;
            RadarPara.NrTx = generateRangeAngle ? nbrTxElements : 1;
            RadarPara.TxSpacing = 2. * lambda;

   /* Parameters relevant to output results */
   RadarPara.WindFunc = WinProp_FMCW_Para::Windowing::HANNING;
   RadarPara.GenerateRangeDoppler = true;
   RadarPara.GenerateRangeAngle = generateRangeAngle;
   RadarPara.GenerteIQ = true;
            sprintf(RadarPara.OutputPathIQ, "%s%s" , API_DATA_FOLDER, "output/
indoor_propagation_fmcw/Radar_IQ_data.txt");

   /* Range-Doppler heat map result */
   WinProp_ResultPlane RangeVelocity;
   WinProp_Structure_Init_ResultPlane(&RangeVelocity);

   /* Range-Angle heat map result */
   WinProp_ResultPlane RangeAngle;
   WinProp_Structure_Init_ResultPlane(&RangeAngle);

   /*------- FMCW radar post-processing -------*/
   WinProp_FMCW_Radar FMCW_Radar;
   Error = FMCW_Radar.compute(&RadarPara,
               generateRangeAngle ? RadarPara.NrRx * RadarPara.NrTx : 1,
    generateRangeAngle ? subStreamRays : &PowerResult->ResultPoints[0].Rays,
    &WinPropCallback,
    &RangeVelocity,
```

```
    generateRangeAngle ? &RangeAngle : nullptr);

   /* Use RangeVelocity to find the range and velocity of the targets */

   if (Error == 0 && RadarPara.GenerateRangeDoppler)
   {
    /* Save values in ASCII file */
    const char* Filename = API_DATA_FOLDER "output/indoor_propagation_fmcw/
rangeDopplerHeatmap.txt";
    FILE* OutputFile = fopen(Filename, "w");
    for (int i = 0; i < RangeVelocity.Columns; i++)
    {
     for (int j = 0; j < RangeVelocity.Lines; j++)
     {
      fprintf(OutputFile, "%.3f\t%.3f\t%.3f\n",
       RangeVelocity.Matrix[i][j].Location.x,
       RangeVelocity.Matrix[i][j].Location.y,
       RangeVelocity.Matrix[i][j].Value);
     }
    }

    /* Close file. */
    fclose(OutputFile);
   }

   if (Error == 0 && RadarPara.GenerateRangeAngle)
   {
    /* Save values in ASCII file */
    const char* Filename = API_DATA_FOLDER "output/indoor_propagation_fmcw/
rangeAngleHeatmap.txt";
    FILE* OutputFile = fopen(Filename, "w");
    for (int i = 0; i < RangeAngle.Columns; i++)
    {
     for (int j = 0; j < RangeAngle.Lines; j++)
     {
      fprintf(OutputFile, "%.3f\t%.3f\t%.3f\n",
       RangeAngle.Matrix[i][j].Location.x,
       RangeAngle.Matrix[i][j].Location.y,
       RangeAngle.Matrix[i][j].Value);
     }
    }

    /* Close file. */
    fclose(OutputFile);
   }

   /* Free memory */
   WinProp_Structure_Free_ResultPlane(&RangeVelocity);
   WinProp_Structure_Free_ResultPlane(&RangeAngle);
  }

  /* Free memory */
  WinProp_Structure_Free_ResultPointsList(PowerResult);

  for (int i = 0; i < nbrElements; i++)
   WinProp_FreeRayMatrixElement(&subStreamRays[i]);

  /* Close project */
  WinProp_Close(ProjectHandle);

 }
```

```
 return 0;
}

int _STD_CALL CallbackMessage(const char * Text)
{
 if (Text == nullptr)
  return 0;

 std::cout << "\n" << Text;

 return(0);
}

int _STD_CALL CallbackError(const char * Text, int Error)
{
 if (Text == nullptr)
  return 0;

 std::cout << "\n";

#ifdef __LINUX
 std::cout << "\033[31m" << "Error (" << Error << "): "; // highlight error in red
 color
#else
 HANDLE hConsole = GetStdHandle(STD_OUTPUT_HANDLE);
 SetConsoleTextAttribute(hConsole, FOREGROUND_RED);
 std::cout << "Error (" << Error << "): ";
#endif // __LINUX
 std::cout << Text;

#ifdef __LINUX
 std::cout << "\033[0m"; // highlight error in red color
#else
 SetConsoleTextAttribute(hConsole, FOREGROUND_RED | FOREGROUND_BLUE |
 FOREGROUND_GREEN);
#endif // __LINUX

 return 0;
}

int _STD_CALL CallbackProgress(int value, const char* text)
{
 char Line[200];

 sprintf(Line, "\n%d%% %s", value, text);
 std::cout << Line;

 return(0);
}

// Helper functions
void write_ascii(const WinProp_ResultPointsList * Resultmatrix, const char* Filename)
{
 FILE* OutputFile = fopen(Filename,"w");
 if (OutputFile)
 {
  for (int p = 0; p < Resultmatrix->NrResultPoints; p++)
  {
   double result = Resultmatrix->ResultPoints[p].ResultValue;
   double Coordinate_X = Resultmatrix->ResultPoints[p].Location.x;
   double Coordinate_Y = Resultmatrix->ResultPoints[p].Location.y;
```

```
   double Coordinate_Z = Resultmatrix->ResultPoints[p].Location.z;

   fprintf(OutputFile, "%.2f\t%.2f\t%.2f\t%.2f\n", Coordinate_X, Coordinate_Y,
 Coordinate_Z, result);
  }

  /* Close file. */
  fclose(OutputFile);
 }
 else
  printf("\nCould not open the File: %s for writing.\n",Filename);
}
```

2.7 WinProp_FMCW_Radar

Function List

WinProp_FMCW_Radar::WinProp_FMCW_Radar
: Default constructor

WinProp_FMCW_Radar::~WinProp_FMCW_Radar
: Destructor

WinProp_FMCW_Radar::compute
: Computes the Range-Doppler and/or Range-Angle heat maps using a FMCW radar by post-processing the received rays with sawtooth chirps according to the radar parameters.

Function Details

WinProp_FMCW_Radar()

Description

Default constructor

Parameters

Returns None

~WinProp_FMCW_Radar()

Description

Destructor

Parameters

Returns None

int **compute**(*const* WinProp_FMCW_Para * *fmcwPara, const int nrRayMatrices, const* WinProp_RayMatrixElement * *rayMatrices, const* WinProp_Callback * *callbacks,* WinProp_ResultPlane * *rangeVelocity,* WinProp_ResultPlane * *rangeAngle)*

Description

Computes the Range-Doppler and/or Range-Angle heat maps using a FMCW radar by post-processing the received rays with sawtooth chirps according to the radar parameters.

Parameters

const WinProp_FMCW_Para * *fmcwPara*
: The FMCW radar parameters.

const int nrRayMatrices
: The number of ray matrices.

const WinProp_RayMatrixElement * *rayMatrices*
: The ray matrices. One ray matrix for Range-Doppler heat map result for one time step. Multiple ray matrices from multiple transmitting and receiving elements for angle estimation relevant for WinProp_FMCW_Para::ElementsConfig::MULTIPLE_ELEMENTS. Multiple

ray matrices from multiple time steps, supporting time-series analysis relevant for WinProp_FMCW_Para::ElementsConfig::MULTIPLE_TIME_STEPS.

const WinProp_Callback * *callbacks*
If non-null, the callbacks to be used.

WinProp_ResultPlane * *rangeVelocity*
If non-null, structure containing the Range-Doppler heat map result.

WinProp_ResultPlane * *rangeAngle*
If non-null, structure containing the Range-Angle heat map result.

Returns An integer: 0 = success, failure otherwise.

The documentation was generated from the following file:

- `source/FMCW/winprop_fmcw_radar.h`

2.8 Beam Generation

Function List

WinProp_GenerateBeams
: Generates patterns of individual beams and the corresponding envelope pattern from a set of parameters. It returns optionally (if non-null) the envelope patterns for control/data beams.

WinProp_ConvertBeams
: Converts/combines a set of individual beams that are available in a set of individual antenna-pattern files into envelope and individual beam patterns. It returns optionally (if non-null) the envelope patterns for control/data beams.

WinProp_GenerateAntennaArrayPattern
: Generates a beam radiation pattern using linear/planar phased array antenna. It returns optionally (if non-null) the beam pattern.

Function Details

int ***WinProp_GenerateBeams****(const* WinProp_Generate_Beams * *parameters, const* WinProp_Callback * *callbacks,* WinProp_Pattern * *envelopeCtrl,* WinProp_Pattern * *envelopeData)*

Description

Generates patterns of individual beams and the corresponding envelope pattern from a set of parameters. It returns optionally (if non-null) the envelope patterns for control/data beams.

Parameters

const WinProp_Generate_Beams * *parameters*
: Parameters for controlling the operation. See WinProp_Generate_Beams.

const WinProp_Callback * *callbacks*
: If non-null, the callbacks to be used.

WinProp_Pattern * *envelopeCtrl*
: If non-null, structure containing the envelope pattern for control beams.

WinProp_Pattern * *envelopeData*
: If non-null, structure containing the envelope pattern for data beams.

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_ConvertBeams****(const* WinProp_Convert_Beams * *parameters, const* WinProp_Callback * *callbacks,* WinProp_Pattern * *envelopeCtrl,* WinProp_Pattern * *envelopeData)*

Description

Converts/combines a set of individual beams that are available in a set of individual antenna-pattern files into envelope and individual beam patterns. It returns optionally (if non-null) the envelope patterns for control/data beams.

Parameters

const WinProp_Convert_Beams * *parameters*
Parameters for controlling the operation. See WinProp_Convert_Beams.

const WinProp_Callback * *callbacks*
If non-null, the callbacks to be used.

WinProp_Pattern * *envelopeCtrl*
If non-null, structure containing the envelope pattern for control beams.

WinProp_Pattern * *envelopeData*
If non-null, structure containing the envelope pattern for data beams.

Returns An integer: 0 = success, failure otherwise.

int ***WinProp_GenerateAntennaArrayPattern****(const* WinProp_Antenna_Array * *antennaArray, const* WinProp_Callback * *callbacks,* WinProp_Pattern * *beamPattern)*

Description

Generates a beam radiation pattern using linear/planar phased array antenna. It returns optionally (if non-null) the beam pattern.

Parameters

const WinProp_Antenna_Array * *antennaArray*
Antenna array configuration. See WinProp_Antenna_Array.

const WinProp_Callback * *callbacks*
If non-null, the callbacks to be used.

WinProp_Pattern * *beamPattern*
If non-null, structure containing the beam pattern.

Returns An integer: 0 = success, failure otherwise.

The documentation was generated from the following file:

- `source/Beams/winprop_beams_main.h`

2.9 Beam type defines

Detailed Description

Enum

WINPROP_ARRAY_ELEMENT_TYPE

- ARRAY_ELEMENT_ISOTROPIC : Isotropic antenna element. See WinProp_Antenna_Array::BaseElement where this is used.
- ARRAY_ELEMENT_DIPOLE : Dipole antenna element. See WinProp_Antenna_Array::BaseElement where this is used.
- ARRAY_ELEMENT_EXTERNAL : External antenna element. See WinProp_Antenna_Array::BaseElement where this is used.

WINPROP_ARRAY_AMPLITUE_TAPER

- ARRAY_AMPLITUE_TAPER_UNIFORM : Unifrom amplitudes. See WinProp_Antenna_Array::AmplitudeTaper where this is used.
- ARRAY_AMPLITUE_TAPER_TRAINGLE : Triangular amplitude taper. See WinProp_Antenna_Array::AmplitudeTaper where this is used.
- ARRAY_AMPLITUE_TAPER_COSINE : Cosine amplitude taper. See WinProp_Antenna_Array::AmplitudeTaper where this is used.
- ARRAY_AMPLITUE_TAPER_DOLPH_CHEBYSHEV : Dolph-Chebyshev amplitude taper. See WinProp_Antenna_Array::AmplitudeTaper where this is used.

WINPROP_ARRAY_PLANE

- ARRAY_PLANE_XY : XY-Plane. See WinProp_Antenna_Array::AntennaArrayPlane where this is used.
- ARRAY_PLANE_XZ : XZ-Plane. See WinProp_Antenna_Array::AntennaArrayPlane where this is used.

The documentation was generated from the following file:

- `source/Beams/winprop_beams_types.h`

2.10 File name defines

Detailed Description

Other defines

typedef char WinProp_Filename[512]
: define a type for file names and file paths

typedef char WinProp_NameString[150]
: define a type for names

typedef char WinProp_ErrorString[300]
: define a type for names

The documentation was generated from the following file:

- `source/Interface/WP_Filename.h`

2.11 Additional callback defines

Detailed Description

Other defines

_STD_CALL *__stdcall*

The documentation was generated from the following file:

- `source/Shared/Callback/CallbackDefines.h`

2.12 Error codes

Detailed Description

Error codes returned by the WinProp API.

Error messages

WINPROP_ERROR_NULLPOINTER *1*
: Null pointer returned.

WINPROP_ERROR_SCENARIODATAMISSING *2*
: Missing scenario data.

WINPROP_ERROR_MEMORY *3*
: Memory related error.

WINPROP_ERROR_LISTFULL *4*
: Number of predictions exceeded.

WINPROP_ERROR_DATABASENAME *5*
: Error when reading/writing database.

WINPROP_ERROR_PROJECTNOTFOUND *6*
: Project not found.

WINPROP_ERROR_RESULTALLOCATION *7*
: Error during allocation of result matrix for point/trajectory mode results.

WINPROP_ERROR_PREDICTION *8*
: Error during prediction based on measurements.

WINPROP_ERROR_BUILDING *9*
: Error during indoor building initialisation.

WINPROP_ERROR_VECTORDATABASE *10*
: Error during database conversion to the WinProp format.

WINPROP_ERROR_NOTSUPPORTED *11*
: Operation not supported.

WINPROP_ERROR_ERRORNOTFOUND *12*
: Error message not found.

WINPROP_ERROR_LICENSENOTVALID *13*
: Invalid license.

WINPROP_ERROR_ENGINEBUSY *14*
: Returned when an operation is attempted while prediction engine is busy in the middle of a computation.

WINPROP_ERROR_NOTCALIBRATED *15*
: Calibration not yet done.

WINPROP_ERROR_WRONGRECORDSIZE *16*
: Error when the size of calibration data or the calibration version is incorrect.

WINPROP_ERROR_UNIT *17*
: Incorrect data type specified. See WinProp_Result::DataType.

WINPROP_ERROR_WRITERESULT *18*
: Error when writing results.

WINPROP_ERROR_VERSIONUNKNOWN *19*
: Error when determining the version or release date.

WINPROP_ERROR_ANTENNAPATTERN *20*
: Error when the defined parameters of the antenna pattern (see WinProp_Pattern) are inconsistent.

WINPROP_ERROR_FILTER *21*
: Error during filtering of results. This may be returned by WinProp_FilterResultPlanes or WinProp_FilterResult.

WINPROP_ERROR_ANTENNATYPE *22*
: Unknown antenna type defined at WinProp_Antenna::Type.

WINPROP_ERROR_CONVERTER *23*
: Error during database conversion. This may be returned by WinProp_Convert.

WINPROP_ERROR_AUTOCAD_OPEN_FILE *24*
: Error when converting AutoCAD file formats (.dxf, .dwg) into WinProp's format.

WINPROP_ERROR_TRANSMITTER_INSIDE_WALL *25*
: Transmitter is located inside a wall.

WINPROP_ERROR_COMPUTE_RESULT *26*
: Error during network planning. This may be returned by WinProp_Net_Project_Compute.

WINPROP_ERROR_INDEX_NOT_VALID *27*
: Error when adding a new MCS/transmission mode/service for network planning. This may be returned by WinProp_Net_TransmissionMode_Add.

WINPROP_ERROR_TRANSMITTER_ALREADY_DEFINED *28*
: Transmitter duplicate found.

WINPROP_ERROR_INTERNAL_ERROR *29*
: Internal error issued for an undefined state.

WINPROP_ERROR_NOT_ENOUGH_DATA *30*
: Not enough measurement points defined at WinProp_Surveydata::NbrValues.

WINPROP_ERROR_HEIGHT_NOT_PREPROCESSED *31*
: The user defined prediction height is not available in the list of pre-processed heights of a database.

WINPROP_ERROR_RESOLUTION_NOT_PREPROCESSED *32*
: The user defined resolution does not match the available pre-processed resolution(s) in a database.

WINPROP_ERROR_PREDICTION_PLANE_WRONG *33*
: Prediction plane is incorrectly defined.

WINPROP_ERROR_FEATURE_NOT_LICENSED *34*
: The feature is not supported by the current license.

WINPROP_ERROR_TRANSMITTER_OUTSIDE_TOPO *35*
: The transmitter is located outside the topography.

WINPROP_ERROR_RECEIVER_OUTSIDE_TOPO *36*
: The receiver is located outside the topography.

WINPROP_ERROR_VECTORS_OUTSIDE_TOPO *37*
: Walls in a database are located outside the topographical area.

WINPROP_ERROR_TX_GROUP_NOT_IN_DATABASE *38*
: Transmitter not defined in the database.

WINPROP_ERROR_TX_MOVING_NO_MEMORY *39*
: Insufficient memory to during transmitter initialisation in a database with moving objects.

WINPROP_ERROR_TX_MOVING_BASED_ON_GROUP *40*
: Error during initialisation of a database with moving objects.

WINPROP_ERROR_LICENSENOTSUPPORTED *41*
: License does not support the requested feature.

WINPROP_ERROR_INVALID_ARGUMENT *42*
: Invalid argument defined.

WINPROP_ERROR_READRESULT *43*
: Error while reading results.

WINPROP_ERROR_ANTENNA *44*
: Error with the defined antenna parameters (see WinProp_Antenna).

WINPROP_ERROR_WRONG_ANTENNA_PATTERN *45*
: Non-ffe antenna pattern for full polarimetric project.

WINPROP_ERROR_TX_DYNAMIC_SET_NOT_DEFINED_FOR_STEP *46*
: Error for defined time variant sets for moving transmitter.

WINPROP_ERROR_TX_MOVING_BASED_ON_TRAJECTORY *47*
Error while creating dynamics for a moving transmitter along a trajectory.

The documentation was generated from the following file:

- `source/Interface/Errors.h`

2.13 Defines of ray paths.

Detailed Description

Enum

WINPROP_RAY_INTERACTION_TYPE

- RAY_INTERACTION_UNKOWN : Unknown interaction.
- RAY_INTERACTION_REFLECTION : Ray reflection.
- RAY_INTERACTION_DIFFRACTION : Ray diffraction.
- RAY_INTERACTION_DIFFRACTION_KE : Ray knife-edge diffraction.
- RAY_INTERACTION_SCATTERING : Ray scattering.
- RAY_INTERACTION_SCATTERING_RCS : Ray scattering at a RCS object.
- RAY_INTERACTION_TRANSMISSION : Ray transmission.
- RAY_INTERACTION_RECEIVER : Ray interaction at receiver.

WINPROP_RAY_PIXEL_TYPE

- RAY_PIXEL_TYPE_UNKOWN : Unknown pixel type.
- RAY_PIXEL_TYPE_STANDARD : Standard pixel.
- RAY_PIXEL_TYPE_INTERPOLATED : Pixel is interpolated.
- RAY_PIXEL_TYPE_INDOOR : Pixel is within a building in an urban scenario.

The documentation was generated from the following file:

- `source/Interface/Paths.h`

2.14 Ellipsoid coordinate systems

Detailed Description

Definition of supported ellipsoids.

COORD_SYSTEM_ELLIPSOID_UNKNOWN *-1*
: Unknown ellipsoid.

COORD_SYSTEM_ELLIPSOID_WGS_84 *1*
: World Geodetic System 84 ellipsoid.

COORD_SYSTEM_ELLIPSOID_WGS_72 *2*
: World Geodetic System 72 ellipsoid.

COORD_SYSTEM_ELLIPSOID_WGS_66 *3*
: World Geodetic System 66 ellipsoid.

COORD_SYSTEM_ELLIPSOID_WGS_60 *4*
: World Geodetic System 60 ellipsoid.

COORD_SYSTEM_ELLIPSOID_SOUTH_AMERICAN *5*
: South American ellipsoid.

COORD_SYSTEM_ELLIPSOID_KRASSOVSKY *6*
: Krassovsky ellipsoid.

COORD_SYSTEM_ELLIPSOID_HOUGH *7*
: Hough ellipsoid.

COORD_SYSTEM_ELLIPSOID_HELMERT *8*
: Helmert ellipsoid.

COORD_SYSTEM_ELLIPSOID_AUSTRALIAN_NATIONAL *9*
: Australian national ellipsoid.

COORD_SYSTEM_ELLIPSOID_AIRY *10*
: Airy ellipsoid.

COORD_SYSTEM_ELLIPSOID_MODIFIED_AIRY *11*
: Modified Airy ellipsoid.

COORD_SYSTEM_ELLIPSOID_BESSEL_1841 *12*
: Bessel 1841 ellipsoid.

COORD_SYSTEM_ELLIPSOID_BESSEL_1841_NAMBIA *13*
: Bessel 1841 Namibia ellipsoid.

COORD_SYSTEM_ELLIPSOID_CLARKE_1866 *14*
Clarke 1866 ellipsoid.

COORD_SYSTEM_ELLIPSOID_CLARKE_1880 *15*
Clarke 1880 ellipsoid.

COORD_SYSTEM_ELLIPSOID_FISCHER_1960 *16*
Fischer 1960 ellipsoid.

COORD_SYSTEM_ELLIPSOID_FISCHER_1968 *17*
Fischer 1968 ellipsoid.

COORD_SYSTEM_ELLIPSOID_FISCHER_MODIFIED *18*
Modified Fischer ellipsoid.

COORD_SYSTEM_ELLIPSOID_GRS_1967 *19*
GRS 1967 ellipsoid.

COORD_SYSTEM_ELLIPSOID_GRS_1980 *20*
GRS 1980 ellipsoid.

COORD_SYSTEM_ELLIPSOID_INTERNATIONAL *21*
International ellipsoid.

COORD_SYSTEM_ELLIPSOID_EVEREST *22*
Everest ellipsoid.

COORD_SYSTEM_ELLIPSOID_EVEREST_BRUNEI *23*
Everest ellipsoid - Brunei.

COORD_SYSTEM_ELLIPSOID_EVEREST_INDIA *24*
Everest ellipsoid - India.

COORD_SYSTEM_ELLIPSOID_EVEREST_MALASIA *25*
Everest ellipsoid - Malaysia.

COORD_SYSTEM_ELLIPSOID_EVEREST_SING *26*
Everest ellipsoid - Singapore.

COORD_SYSTEM_ELLIPSOID_EVEREST_PAKISTAN *27*
Everest ellipsoid - Pakistan.

COORD_SYSTEM_ELLIPSOID_INDONESIAN *28*
Everest ellipsoid - Indonesia.

Other defines

COORD_SYSTEM_ELLIPSOID_NUMBER *29*
the total number of supported ellipsoids.

The documentation was generated from the following file:

- `source/Shared/Coordinates/EllipsoidID.h`

2.15 Data Structures

Here is a list of all data structures with brief descriptions

2.15.1 AC_LayerInfo

Data Fields

char
layerName [400]

char
unitString [400]

int
option

double
height2dAboveGroundPlane

double
height2dGroundPlane

int
askNoMore

Detailed Description

Input for the CALLBACK_AC_CONVERTER callback for gathering settings of how to convert found 2D objects during conversion of AutoCAD files (dwg, dxf).

Field Documentation

layerName

char ***AC_LayerInfo::layerName****[400]*
Name of the AutoCAD layer (for display only).

unitString

char ***AC_LayerInfo::unitString****[400]*
Current measurement unit (for display only).

option

int ***AC_LayerInfo::option***
Conversion option.

- 0 = Set height of all 2D objects.
- 1 = Ignore all 2D objects.

height2dAboveGroundPlane

double ***AC_LayerInfo::height2dAboveGroundPlane***
Set height of 2D object relative to ground.

height2dGroundPlane

double ***AC_LayerInfo::height2dGroundPlane***
Set height of ground plane.

askNoMore

int ***AC_LayerInfo::askNoMore***
If set to a value not equal to 0, the settings are applied for all subsequent layers.

The documentation was generated from the following file:

- `source/Interface/EngineConvert.h`

2.15.2 CLUTTER

Data Fields

unsigned int
columns

unsigned int
rows

double
resolutionX

double
resolutionY

double
lowerLeftX

double
lowerLeftY

double
upperRightX

double
upperRightY

int **
clutterData

int
nbrClasses

CLUTTER_CLASS *
classes

Detailed Description

Clutter definition.

Field Documentation

columns

unsigned int ***CLUTTER::columns***
columns of clutter pixel matrix

rows

unsigned int ***CLUTTER::rows***
rows of clutter pixel matrix

resolutionX

double ***CLUTTER::resolutionX***
resolution of clutter pixel matrix

resolutionY

double ***CLUTTER::resolutionY***
resolution of clutter pixel matrix

lowerLeftX

double ***CLUTTER::lowerLeftX***
lower left corner of clutter pixel matrix

lowerLeftY

double ***CLUTTER::lowerLeftY***
lower left corner of clutter pixel matrix

upperRightX

double ***CLUTTER::upperRightX***
upper right corner of clutter pixel matrix

upperRightY

double ***CLUTTER::upperRightY***
upper right corner of clutter pixel matrix

clutterData

*int ** ***CLUTTER::clutterData***
pixel matrix with corresponding clutter ID

nbrClasses

int ***CLUTTER::nbrClasses***
number of clutter classes

classes

CLUTTER_CLASS * ***CLUTTER::classes***
array with clutter classes

The documentation was generated from the following file:

- `source/Public/Interface/InterfaceType.h`

2.15.3 CLUTTER_CLASS

Data Fields

char
name [CLUTTER_CLASS_NAME_LENGTH]

int
hataClass

int
ID

int
NrAttenuation

FREQUENCY_LOSS *
Attenuation

float
heightMean

float
heightStdDev

float
clearanceMin

float
clearanceMax

float
clearanceRatio

float
clearanceFree

float
Weight

int
ITU_1546

int
isVegetation

int
PixelCount

float
MeanOffset

unsigned char
colorRed

unsigned char
colorGreen

unsigned char
colorBlue

Detailed Description

Clutter class definition.

Field Documentation

name

char ***CLUTTER_CLASS::name****[CLUTTER_CLASS_NAME_LENGTH]*
name of clutter class

hataClass

int ***CLUTTER_CLASS::hataClass***
corresponding hata class

ID

int ***CLUTTER_CLASS::ID***
ID of clutter class

NrAttenuation

int ***CLUTTER_CLASS::NrAttenuation***
size of array with losses for frequency

Attenuation

FREQUENCY_LOSS * ***CLUTTER_CLASS::Attenuation***
array with losses [dB] for frequency

heightMean

float ***CLUTTER_CLASS::heightMean***
(mean) height of objects

heightStdDev

float ***CLUTTER_CLASS::heightStdDev***
std. dev. of heights

clearanceMin

float ***CLUTTER_CLASS::clearanceMin***
clearance (min) in meters

clearanceMax

float ***CLUTTER_CLASS::clearanceMax***
clearance (max) in meters

clearanceRatio

float ***CLUTTER_CLASS::clearanceRatio***
clearance ratio (in percent)

clearanceFree

float ***CLUTTER_CLASS::clearanceFree***
clearance (open area) in meters

Weight

float ***CLUTTER_CLASS::Weight***
weight for dominant class search

ITU_1546

int ***CLUTTER_CLASS::ITU_1546***
spec accord. to ITU 1546

isVegetation

int ***CLUTTER_CLASS::isVegetation***
Consider as vegetation (yes/no)

PixelCount

int ***CLUTTER_CLASS::PixelCount***
Number of pixels

MeanOffset

float ***CLUTTER_CLASS::MeanOffset***
Mean offset.

colorRed

unsigned char ***CLUTTER_CLASS::colorRed***

colorGreen

unsigned char ***CLUTTER_CLASS::colorGreen***

colorBlue

unsigned char ***CLUTTER_CLASS::colorBlue***

The documentation was generated from the following file:

- `source/Public/Interface/InterfaceType.h`

2.15.4 COMPLEX_VALUE

Data Fields

double
Re

double
Im

Detailed Description

A Complex number

Field Documentation

Re

double ***COMPLEX_VALUE::Re***
Real part

Im

double ***COMPLEX_VALUE::Im***
Imaginary part

The documentation was generated from the following file:

- `source/Interface/Complex.h`

2.15.5 COORDPOINT

Data Fields

double
x

double
y

double
z

Detailed Description

Cartesian coordinates of point in 3D space

Field Documentation

x

*double **COORDPOINT::x***
The x coordinate

y

*double **COORDPOINT::y***
The y coordinate

z

*double **COORDPOINT::z***
The z coordinate

The documentation was generated from the following file:

- `source/Interface/CoordPoint.h`

2.15.6 FIELD_VECTORS

Data Fields

COMPLEX_VALUE
PolVfieldX

COMPLEX_VALUE
PolVfieldY

COMPLEX_VALUE
PolVfieldZ

COMPLEX_VALUE
PolHfieldX

COMPLEX_VALUE
PolHfieldY

COMPLEX_VALUE
PolHfieldZ

Detailed Description

Field Documentation

PolVfieldX

COMPLEX_VALUE ***FIELD_VECTORS::PolVfieldX***
Complex value of linear field strength (uV/m) of x component for vertical polarization

PolVfieldY

COMPLEX_VALUE ***FIELD_VECTORS::PolVfieldY***
Complex value of linear field strength (uV/m) of y component for vertical polarization

PolVfieldZ

COMPLEX_VALUE ***FIELD_VECTORS::PolVfieldZ***
Complex value of linear field strength (uV/m) of z component for vertical polarization

PolHfieldX

COMPLEX_VALUE ***FIELD_VECTORS::PolHfieldX***
Complex value of linear field strength (uV/m) of x component for horizontal polarization

PolHfieldY

COMPLEX_VALUE ***FIELD_VECTORS::PolHfieldY***
Complex value of linear field strength (uV/m) of y component for horizontal polarization

PolHfieldZ

COMPLEX_VALUE ***FIELD_VECTORS::PolHfieldZ***
Complex value of linear field strength (uV/m) of z component for horizontal polarization

The documentation was generated from the following file:

- `source/Interface/WP_Result.h`

2.15.7 FREQUENCY_LOSS

Data Fields

float
FrequencyMin_MHz

float
FrequencyMax_MHz

float
Loss_dB

float
AttenHataDense

float
AttenHataMedium

float
AttenHataSuburban

float
AttenHataOpen

float
Dielectricity

float
Permeability

float
Conductivity

Detailed Description

Electrical properties

Field Documentation

FrequencyMin_MHz

float ***FREQUENCY_LOSS::FrequencyMin_MHz***
Min. Frequency in MHz of band

FrequencyMax_MHz

float ***FREQUENCY_LOSS::FrequencyMax_MHz***
Max. Frequency in MHz of band

Loss_dB

float ***FREQUENCY_LOSS::Loss_dB***
Loss in dB for the given frequency band.

AttenHataDense

float ***FREQUENCY_LOSS::AttenHataDense***
Attenuation of Hata dense urban

AttenHataMedium

float ***FREQUENCY_LOSS::AttenHataMedium***
Attenuation Hata medium urban

AttenHataSuburban

float ***FREQUENCY_LOSS::AttenHataSuburban***
Attenuation of Hata suburban

AttenHataOpen

float ***FREQUENCY_LOSS::AttenHataOpen***
Attenuation of Hata open area

Dielectricity

float ***FREQUENCY_LOSS::Dielectricity***
Dielectricity.

Permeability

float ***FREQUENCY_LOSS::Permeability***
Permeability.

Conductivity

float ***FREQUENCY_LOSS::Conductivity***
Conductivity.

The documentation was generated from the following file:

- `source/Public/Interface/InterfaceType.h`

2.15.8 INDOOR_WALL

Data Fields

int
NrCorners

COORDPOINT *
Corners

int
MaterialIndex

Detailed Description

Indoor wall definition.

Field Documentation

NrCorners

int ***INDOOR_WALL::NrCorners***
Number of corners.

Corners

COORDPOINT * ***INDOOR_WALL::Corners***
Corner coordinates.

MaterialIndex

int ***INDOOR_WALL::MaterialIndex***
Material index.

The documentation was generated from the following file:

- source/Public/Interface/IndoorWalls.h

2.15.9 INDOOR_WALLS

Data Fields

int
NrWalls

INDOOR_WALL *
Walls

Detailed Description

List with indoor walls

Field Documentation

NrWalls

int ***INDOOR_WALLS::NrWalls***
Number of walls.

Walls

INDOOR_WALL * ***INDOOR_WALLS::Walls***
Walls.

The documentation was generated from the following file:

- source/Public/Interface/IndoorWalls.h

2.15.10 INTERFACE_CORNER

Data Fields

double
x

double
y

double
z

Detailed Description

An interface corner.

Field Documentation

x

double ***INTERFACE_CORNER::x***
x-coordinates of corner

y

double ***INTERFACE_CORNER::y***
y-coordinates of corner

z

double ***INTERFACE_CORNER::z***
z-coordinates of corner

The documentation was generated from the following file:

- `source/Public/Interface/InterfaceType.h`

2.15.11 INTERFACE_MATERIAL

Data Fields

int
materialID

float
thickness

float
epsilon

float
mu

float
sigma

float
transWall

float
reflexion

float
diffInMin

float
diffInMax

float
diffOut

float
scatteringLoss30

float
SurfaceRoughness

Detailed Description

Interface material definition.

Field Documentation

materialID

int ***INTERFACE_MATERIAL::materialID***
unique ID of material

thickness

float ***INTERFACE_MATERIAL::thickness***
thickness of wall

epsilon

float ***INTERFACE_MATERIAL::epsilon***
Permittivity.

mu

float ***INTERFACE_MATERIAL::mu***
Permeability.

sigma

float ***INTERFACE_MATERIAL::sigma***
Conductivity.

transWall

float ***INTERFACE_MATERIAL::transWall***
Transmission loss of wall [dB].

reflexion

float ***INTERFACE_MATERIAL::reflexion***
Reflexion loss [dB].

diffInMin

float ***INTERFACE_MATERIAL::diffInMin***
Minimum diffraction loss.

diffInMax

float ***INTERFACE_MATERIAL::diffInMax***
Maximum diffraction loss.

diffOut

float ***INTERFACE_MATERIAL::diffOut***
Diffraction loss.

scatteringLoss30

float ***INTERFACE_MATERIAL::scatteringLoss30***
Scatter loss at 30Â° angle diff.

SurfaceRoughness

float ***INTERFACE_MATERIAL::SurfaceRoughness***
Surface roughness in cm.

The documentation was generated from the following file:

- `source/Public/Interface/InterfaceType.h`

2.15.12 MATERIAL

Data Fields

int
Index

char
Name [300]

int
NrBands

MATERIALBAND *
Bands

unsigned char
ColorRed

unsigned char
ColorGreen

unsigned char
ColorBlue

Detailed Description

Material properties of indoor walls

Field Documentation

Index

int ***MATERIAL::Index***
Wall index.

Name

char ***MATERIAL::Name****[300]*
Name.

NrBands

int ***MATERIAL::NrBands***
Number of bands.

Bands

MATERIALBAND * ***MATERIAL::Bands***
Material properties.

ColorRed

unsigned char ***MATERIAL::ColorRed***

ColorGreen

unsigned char ***MATERIAL::ColorGreen***

ColorBlue

unsigned char ***MATERIAL::ColorBlue***

The documentation was generated from the following file:

- `source/Public/Interface/InterfaceMaterial.h`

2.15.13 MATERIALBAND

Data Fields

double
Epsilon

double
Mu

double
Sigma

double
ScatteringVV

double
ScatteringVH

double
ScatteringHV

double
ScatteringHH

double
TransmissionLoss

double
ReflectionLoss

double
DiffractionLossInMin

double
DiffractionLossInMax

double
DiffractionLossOut

double
ScatteringLoss30

double
Frequency

Detailed Description

Frequency dependant material properties

Field Documentation

Epsilon

double ***MATERIALBAND::Epsilon***
Permittivity.

Mu

double ***MATERIALBAND::Mu***
Permeability.

Sigma

double ***MATERIALBAND::Sigma***
Conductivity.

ScatteringVV

double ***MATERIALBAND::ScatteringVV***
Scattering factor vv component.

ScatteringVH

double ***MATERIALBAND::ScatteringVH***
Scattering factor vh component.

ScatteringHV

double ***MATERIALBAND::ScatteringHV***
Scattering factor hv component.

ScatteringHH

double ***MATERIALBAND::ScatteringHH***
Scattering factor hh component.

TransmissionLoss

double ***MATERIALBAND::TransmissionLoss***
Transmission loss.

ReflectionLoss

double ***MATERIALBAND::ReflectionLoss***
Reflection loss.

DiffractionLossInMin

double ***MATERIALBAND::DiffractionLossInMin***
Minimum diffraction loss.

DiffractionLossInMax

double ***MATERIALBAND::DiffractionLossInMax***
Maximum diffraction loss.

DiffractionLossOut

double ***MATERIALBAND::DiffractionLossOut***
Diffraction loss.

ScatteringLoss30

double ***MATERIALBAND::ScatteringLoss30***
Scatter loss at 30Â° angle diff.

Frequency

double ***MATERIALBAND::Frequency***
frequency

The documentation was generated from the following file:

- `source/Public/Interface/InterfaceMaterial.h`

2.15.14 MATERIALS

Data Fields

int
NrMaterials

MATERIAL *
Materials

Detailed Description

Material database

Field Documentation

NrMaterials

int ***MATERIALS::NrMaterials***
Number of materials.

Materials

MATERIAL * ***MATERIALS::Materials***
Material properties of indoor walls.

The documentation was generated from the following file:

- `source/Public/Interface/InterfaceMaterial.h`

2.15.15 Model_COST

Data Fields

double
Exponent

double
ExponentNLOS

int
ReducedTransLoss

double
ReducedFactor

int
Mode2D

double
Distance2D

int
ConsiderAngleTransm

Detailed Description

This structure is used to set the model parameters of the COST 231 Model. This structure can be passed optionally to the WinProp_Predict function.

Field Documentation

Exponent

double ***Model_COST::Exponent***
Path Loss Exponent

ExponentNLOS

double ***Model_COST::ExponentNLOS***
Path Loss Exponent NLOS

ReducedTransLoss

int ***Model_COST::ReducedTransLoss***
Use reduced transmission loss feature. The following values are supported:

- 0 = Feature disabled. The transmission losses of walls are not adapted.
- 1 = Reduction applied to linear transmission loss.
- 2 = Reduction applied to logarithmic transmission loss (dB value).

ReducedFactor

double ***Model_COST::ReducedFactor***
Reduction factor for transmission loss in percent, i.e. 25.0 means 25% reduction.

Mode2D

int ***Model_COST::Mode2D***
Use 2D approach for COST model.

- 1 if used
- 0 otherwise

Distance2D

double ***Model_COST::Distance2D***
Distance in which the 2D approach will be considered. Only relevant if 2D approach is used.See also Mode2D.

ConsiderAngleTransm

int ***Model_COST::ConsiderAngleTransm***
Consider angle dependency for computation of transmission loss.

The documentation was generated from the following file:

- `source/Interface/WP_Model.h`

2.15.16 Model_DetTwoRay

Data Fields

int
DirectRay

int
ReflectedRay

float
ToleranceReflAngle

int
SuperposPhase

int
RO_MaterialMode

double
RO_Dielectricity

double
RO_Conductivity

Detailed Description

Parameter for the deterministic two ray model.

Field Documentation

DirectRay

int ***Model_DetTwoRay::DirectRay***
Mode for computation of direct ray

- 0 = Do not compute
- 1 = Compute

ReflectedRay

int ***Model_DetTwoRay::ReflectedRay***
Mode for computation of reflected ray

- 0 = Do not compute
- 1 = Compute

ToleranceReflAngle

float ***Model_DetTwoRay::ToleranceReflAngle***
Tolerance for specular reflection (angle in deg)

- Min = 0
- Max = 10

SuperposPhase

int ***Model_DetTwoRay::SuperposPhase***
Mode for superposition of rays

- 0 = Uncorrelated
- 1 = Coherent (with phase)

RO_MaterialMode

int ***Model_DetTwoRay::RO_MaterialMode***
Mode for consideration of ground material/properties

- 0 = Default as defined in RO_Dielectricity and RO_Conductivity
- 1 = As defined in clutter map

RO_Dielectricity

double ***Model_DetTwoRay::RO_Dielectricity***
Default ground dielectricity (relative), considered if RO_MaterialMode equal to 0

- Min = 1.0
- Max = 10000.0

RO_Conductivity

double ***Model_DetTwoRay::RO_Conductivity***
Default ground conductivity (in S/m), considered if RO_MaterialMode equal to 0

- Min = 1e-6
- Max = 4e7

The documentation was generated from the following file:

- `source/Interface/WP_Model.h`

2.15.17 Model_DPM

Data Fields

double
DiffrLoss

double
DiffrOffset

double
DiffrBreakpoint

double
ExponentLOS

double
ExponentOLOS

double
ExponentNLOS

double
ExponentLOSafterBP

double
ExponentOLOSafterBP

double
ExponentNLOSafterBP

int
Waveguiding

double
WG_Weighting

double
WG_MaxDistance

int
AdaptiveResolution

double
Offset_LOS_Pixels

double
Offset_NonLOS_Pixels

double
BuildingLossPerMeter

double
BuildingLossDistance

double
BuildingLossFactor

double
BuildingLossMaxFresnel

unsigned char
ConsiderFloorsBuildings

unsigned char
ConsiderFloorsTrxBuilding

unsigned char
PathSearchSameSettings

double
AdditionalHeight

double
AdditionalHeightBuildings

unsigned char
IndoorExponentsSameAsUrban

Detailed Description

This structure is used to set the model parameters of the DPM (Dominant Path Model). This structure can be passed optionally to the WinProp_Predict function.

Field Documentation

DiffrLoss

double ***Model_DPM::DiffrLoss***
Maximum diffraction loss [dB] at the in DiffrBreakpoint set angle, allowed range [2, 30]

DiffrOffset

double ***Model_DPM::DiffrOffset***
Minimal diffraction loss [dB], allowed range [0, 10], must be larger than DiffrLoss

DiffrBreakpoint

double ***Model_DPM::DiffrBreakpoint***
Diffraction angle [deg] from which on the maximal diffraction loss (DiffrLoss) is considered, allowed range [70, 90]

ExponentLOS

double ***Model_DPM::ExponentLOS***
Path loss exponent LOS, allowed range [1, 10]

ExponentOLOS

double ***Model_DPM::ExponentOLOS***
Path loss exponent OLOS, allowed range [1, 10]

ExponentNLOS

double ***Model_DPM::ExponentNLOS***
Path loss exponent NLOS (only indoor), allowed range [1, 10]

ExponentLOSafterBP

double ***Model_DPM::ExponentLOSafterBP***
Path loss exponent LOS after breakpoint (only urban and rural), allowed range [1, 10]

ExponentOLOSafterBP

double ***Model_DPM::ExponentOLOSafterBP***
Path loss exponent OLOS after breakpoint (only urban and rural), allowed range [1, 10]

ExponentNLOSafterBP

double ***Model_DPM::ExponentNLOSafterBP***
Path loss exponent NLOS after breakpoint (only only indoor), allowed range [1, 10]

Waveguiding

int ***Model_DPM::Waveguiding***
Waveguiding, set to 1 to enable, 0 otherwise.

WG_Weighting

double ***Model_DPM::WG_Weighting***
Weighting factor for waveguiding in the range (0,1).See also Waveguiding

WG_MaxDistance

double ***Model_DPM::WG_MaxDistance***
Distance [m] for waveguiding consideration, allowed range [0, 100]See also Waveguiding

AdaptiveResolution

int ***Model_DPM::AdaptiveResolution***
Adaptive resolution factor. A value greater 0 activates the adaptive resolution. Upper limit is 4.

Offset_LOS_Pixels

double ***Model_DPM::Offset_LOS_Pixels***
Offset to be added to all LOS pixels, allowed range [-100, 100]

Offset_NonLOS_Pixels

double ***Model_DPM::Offset_NonLOS_Pixels***
Offset to be added to all Non-LOS pixels, allowed range [-100, 100]

BuildingLossPerMeter

double ***Model_DPM::BuildingLossPerMeter***
Attenuation [dB/m] for path over rooftops, allowed range [0, 10]

BuildingLossDistance

double ***Model_DPM::BuildingLossDistance***
Distance [m] to a roof, a ray needs to travel within to apply BuildingLossPerMeter, allowed range [0.1, 15]

BuildingLossFactor

double ***Model_DPM::BuildingLossFactor***
Factor for switching between more empirical over roof top loss computation and a obstacle within fresnel zone strategy.

- 0.0 = empirical strategy is being used (based on pixels within BuildingLossDistance over a roof)
- 1.0 = strategy based on obstacles within a fresnel zone
- 0.5 = both strategies being equally used

BuildingLossMaxFresnel

double ***Model_DPM::BuildingLossMaxFresnel***
The added loss if an obstacle completely blocks the fresnel zone

ConsiderFloorsBuildings

unsigned char ***Model_DPM::ConsiderFloorsBuildings***
Boolean to use floors in buildings, set to 1 to enable, 0 otherwise. Only relevant for indoor scenarios not required for urban scenarios.

ConsiderFloorsTrxBuilding

unsigned char ***Model_DPM::ConsiderFloorsTrxBuilding***
Boolean to use floors in building with Tx, set to 1 to enable, 0 otherwise. Only relevant for indoor scenarios not required for urban scenarios.

PathSearchSameSettings

unsigned char ***Model_DPM::PathSearchSameSettings***
Set to 1 to consider the set exponents and the set diffraction loss also for the DPM path search.

AdditionalHeight

double ***Model_DPM::AdditionalHeight***
Additional height for pixels within topography in case of prediction on absolute height. Needed to get path traveling across the topography, pixels will not be contained in the results (as they're within the topography). Needs to be larger than zero.

AdditionalHeightBuildings

double ***Model_DPM::AdditionalHeightBuildings***
Additional height for pixels within buildings. Needed to get paths traveling across buildings. If WinProp_ParaMain::BuildingsIndoorCoverage is set to COVERAGEINDOOR_ONROOF, the pixels will be contained in the result. Needs to be larger than zero.

IndoorExponentsSameAsUrban

unsigned char ***Model_DPM::IndoorExponentsSameAsUrban***
Boolean to use the urban exponents also for indoor prediction in CNP Hybrid Urban/Indoor (CNP) projects, set to 1 to enable, 0 otherwise. Only relevant for Hybrid Urban/Indoor (CNP) projects, not required indoor scenarios.

The documentation was generated from the following file:

- `source/Interface/WP_Model.h`

2.15.18 Model_FREESPACE

Data Fields

double
Exponent

double
ExponentNLOS

double
Offset

double
OffsetNLOS

Detailed Description

Structure with parameters for the Freespace Model.

Field Documentation

Exponent

double ***Model_FREESPACE::Exponent***
Path Loss Exponent

ExponentNLOS

double ***Model_FREESPACE::ExponentNLOS***
Path Loss Exponent NLOS

Offset

double ***Model_FREESPACE::Offset***
Offset

OffsetNLOS

double ***Model_FREESPACE::OffsetNLOS***
Offset NLOS

The documentation was generated from the following file:

- `source/Interface/WP_Model.h`

2.15.19 Model_MOTLEYKEENAN

Data Fields

float
VerticalTransmission

float
HorizontalTransmission

Detailed Description

Structure with parameters for the Motley-Keenan Model.

Field Documentation

VerticalTransmission

float ***Model_MOTLEYKEENAN::VerticalTransmission***
Wall Transmission Loss

HorizontalTransmission

float ***Model_MOTLEYKEENAN::HorizontalTransmission***
Ceiling Transmission Loss

The documentation was generated from the following file:

- `source/Interface/WP_Model.h`

2.15.20 Model_RAYTRACING

Data Fields

double
SBR_maxTubeArea

int
SBR_adaptive_maxTubeArea

float
SBR_trxDistTubeArea

int
SBR_scatteringDirections

int
SBR_diffractionDirections

int
SBR_trxDirections

float
SBR_resultCatchFactor

int
SBR_postProcessRaysWithSRT

int
SBR_postProcessRayMaxCombination

int
MaxNrReflDiffr

int
MaxNrDiffractions

int
MaxNrReflections

int
MaxNrTransmissions

int
MaxNrScatterings

double
ResolutionScattering

int
GroundScattering

double
ResolutionGroundScattering

int
GroundScattRCSCoeff

int
GroundReflections

int
GroundDiffractions

double
MinLengthWedges

double
MaxInnerAngleWedges

int
IgnoreAdditionalTransm

int
IgnoreAdditionalDiffr

int
IgnoreAdditionalScatt

int
ComputeAlwaysDirectRay

int
ConsiderAngleTransm

double
PathLossExponent

double
PathLossExponentBeforeLOS

double
PathLossExponentAfterLOS

double
PathLossExponentBeforeNLOS

double
PathLossExponentAfterNLOS

double
PathLossExponentSatellite

int
Superposition

int
ConsiderRCS

int
SRT_AZBMaxObjects

int
IRT_PostprocessRaysWithSRT

int
NrRCS

RCS_PARA *
RCS_Para

Detailed Description

This structure is used to set the model parameters of both 3D Ray Tracing models (IRT, SRT and SBR). This structure can be passed optionally to the WinProp_Predict function. It can be initialized with default values by using WinProp_Structure_Init_Model_RayTracing.

Field Documentation

SBR_maxTubeArea

double ***Model_RAYTRACING::SBR_maxTubeArea***
Max side area of the ray tube. If the ray tube area gets larger, the ray will be split. [m]

SBR_adaptive_maxTubeArea

int ***Model_RAYTRACING::SBR_adaptive_maxTubeArea***
Adaptive ray density. If higher than zero and an antenna pattern is given, ray density will be lowered globally by the specified number of steps (SBR_maxTubeArea will be divided by 2^steps). In antenna directions with a higher gain the ray density will be increased again up to the user set density into the max gain direction.

SBR_trxDistTubeArea

float ***Model_RAYTRACING::SBR_trxDistTubeArea***
Distance after which SBR_maxTubeArea needs to be fulfilled the first time. [m]

SBR_scatteringDirections

int ***Model_RAYTRACING::SBR_scatteringDirections***
Number of directions to investigate

SBR_diffractionDirections

int ***Model_RAYTRACING::SBR_diffractionDirections***
Number of directions to investigate after a diffraction, per degree

SBR_trxDirections

int ***Model_RAYTRACING::SBR_trxDirections***
Number of directions to launch at the transmitter (set to -1 to compute it automatically)

SBR_resultCatchFactor

float ***Model_RAYTRACING::SBR_resultCatchFactor***
Scaling factor for the generated result planes

SBR_postProcessRaysWithSRT

int ***Model_RAYTRACING::SBR_postProcessRaysWithSRT***
Enable (1) or disable (0) SRT post-processing of rays found by the SBR

SBR_postProcessRayMaxCombination

int ***Model_RAYTRACING::SBR_postProcessRayMaxCombination***
Max number of SRT combinations found for a SBR ray to be post-processed

MaxNrReflDiffr

int ***Model_RAYTRACING::MaxNrReflDiffr***
Max. number of reflections and diffractions in one ray

MaxNrDiffractions

int ***Model_RAYTRACING::MaxNrDiffractions***
Max. number of diffractions in one ray

MaxNrReflections

int ***Model_RAYTRACING::MaxNrReflections***
Max. number of reflections in one ray

MaxNrTransmissions

int ***Model_RAYTRACING::MaxNrTransmissions***
Max. number of transmissions in one ray

MaxNrScatterings

int ***Model_RAYTRACING::MaxNrScatterings***
Max. number of scatterings in one ray (SRT, SBR and IRT allows only one maximum scattering).

ResolutionScattering

double ***Model_RAYTRACING::ResolutionScattering***
Tile resolution for scattering. Not relevant for IRT model.

GroundScattering

int ***Model_RAYTRACING::GroundScattering***
Allow ground scattering (in case of separate topography). Set to '1' to enable it. Not relevant for IRT model.

ResolutionGroundScattering

double ***Model_RAYTRACING::ResolutionGroundScattering***
Tile resolution at the ground for ground scattering. Not relevant for IRT model.

GroundScattRCSCoeff

int ***Model_RAYTRACING::GroundScattRCSCoeff***
Compute ground scattering based on the RCS coefficient

GroundReflections

int ***Model_RAYTRACING::GroundReflections***
Allow ground reflections in case of vector databases (*.tdv databases). Set to '1' to enable it. Not relevant for IRT model.

GroundDiffractions

int ***Model_RAYTRACING::GroundDiffractions***
Allow ground diffractions in case of vector databases (*.tdv databases). Set to '1' to enable it. Not relevant for IRT model.

MinLengthWedges

double ***Model_RAYTRACING::MinLengthWedges***
Minimal length of wedges (m) to be considered for diffractions. Not relevant for IRT model.

MaxInnerAngleWedges

double ***Model_RAYTRACING::MaxInnerAngleWedges***
Maximal inner angle of wedges (deg) to be considered for diffractions. Not relevant for IRT model.

IgnoreAdditionalTransm

int ***Model_RAYTRACING::IgnoreAdditionalTransm***
Acceleration for all rays with more than specified reflections: Ignore additional transmissions. Set to the number of reflections for which additional transmissions are allowed, -1 to disable. Not supported by IRT model.

IgnoreAdditionalDiffr

int ***Model_RAYTRACING::IgnoreAdditionalDiffr***
Acceleration for all rays with more than specified reflections: Ignore additional diffractions. Set to the number of reflections for which additional diffractions are allowed, -1 to disable. Not supported by IRT model.

IgnoreAdditionalScatt

int ***Model_RAYTRACING::IgnoreAdditionalScatt***
Acceleration for all rays with more than specified reflections: Ignore additional scattering. Set to the number of reflections for which an additional scattering is allowed, -1 to disable. Not supported by IRT model.

ComputeAlwaysDirectRay

int ***Model_RAYTRACING::ComputeAlwaysDirectRay***
Computation mode of the direct ray from Tx to Rx:

- 0 = With respect to set maximum number of transmissions
- 1 = Compute the direct ray always (even is number of transmissions is higher than allowed)
- 2 = Never compute the direct ray

ConsiderAngleTransm

int ***Model_RAYTRACING::ConsiderAngleTransm***
Consider angle dependency for computation of transmission loss.

PathLossExponent

double ***Model_RAYTRACING::PathLossExponent***
Path loss exponent for the IRT model. Not relevant for SRT & SBR model.

PathLossExponentBeforeLOS

double ***Model_RAYTRACING::PathLossExponentBeforeLOS***
Path Loss Exponent before breakpoint for LOS areas for SRT & SBR model. Not relevant for IRT model.

PathLossExponentAfterLOS

double ***Model_RAYTRACING::PathLossExponentAfterLOS***
Path Loss Exponent after breakpoint for LOS areas for SRT & SBR model. Not relevant for IRT model.

PathLossExponentBeforeNLOS

double ***Model_RAYTRACING::PathLossExponentBeforeNLOS***
Path Loss Exponent before breakpoint for NLOS areas for SRT & SBR model. Not relevant for IRT model.

PathLossExponentAfterNLOS

double ***Model_RAYTRACING::PathLossExponentAfterNLOS***
Path Loss Exponent after breakpoint for NLOS areas for SRT & SBR model. Not relevant for IRT model.

PathLossExponentSatellite

double ***Model_RAYTRACING::PathLossExponentSatellite***
Path Loss Exponent for a satellite transmitter for SRT & SBR model. Not relevant for IRT model.

Superposition

int ***Model_RAYTRACING::Superposition***
Mode for superposition of rays

- 0 = Uncorrelated
- 1 = Coherent (with phase)

ConsiderRCS

int ***Model_RAYTRACING::ConsiderRCS***
Consider the RCS if available, set to 1 to enable, 0 otherwise. Only relevant for indoor scenarios having RCS and using SRT model. Not relevant for IRT model.

SRT_AZBMaxObjects

int ***Model_RAYTRACING::SRT_AZBMaxObjects***
The maximal number of objects, for which the SRT uses a special acceleration method.

IRT_PostprocessRaysWithSRT

int ***Model_RAYTRACING::IRT_PostprocessRaysWithSRT***
Enable (1) or disable (0) SRT post-processing of rays found by the IRT

NrRCS

int ***Model_RAYTRACING::NrRCS***
Number of RCSs defined in the scenarios. Only relevant for indoor scenarios using SRT model. Not relevant for IRT model.

RCS_Para

RCS_PARA * ***Model_RAYTRACING::RCS_Para***
Array of RCS parameters. Only relevant for indoor scenarios using SRT model. Not relevant for IRT model.

The documentation was generated from the following file:

- `source/Interface/WP_Model.h`

2.15.21 Model_RuralITM

Data Fields

double
Time

double
Location

int
VariabilityMode

int
TxSiteCriteria

int
RxSiteCriteria

double
TerrainIrregularity

int
Mode

int
ClimateCode

double
CurvatureRadiusFactor

int
GroundPropertiesMode

double
Conductivity

double
Dielectricity

double
Reliability

double
Confidence

Detailed Description

Parameter for the irregular terrain model (ITM, Longley-Rice) model.

Remark Structure can be initialized with default values by using the function WinProp_Structure_Init_Model_RuralITM.

Field Documentation

Time

double ***Model_RuralITM::Time***
Represents the fraction of time during which the losses are less than the calculated loss., allowed range [0.01 to 0.99]

Location

double ***Model_RuralITM::Location***
Represents the fraction of locations at which the losses are less than the calculated loss, allowed range [0.01 to 0.99]

VariabilityMode

int ***Model_RuralITM::VariabilityMode***
Variability mode

- 0 = single message mode
- 1 = accidental mode
- 2 = mobile mode
- 3 = broadcast mode

TxSiteCriteria

int ***Model_RuralITM::TxSiteCriteria***
Transmitter site criteria describing the care taken at each terminal to assure good propagation conditions

- 0 = random
- 1 = careful
- 2 = very careful

RxSiteCriteria

int ***Model_RuralITM::RxSiteCriteria***
Receiver site criteria describing the care taken at each terminal to assure good propagation conditions

- 0 = random
- 1 = careful
- 2 = very careful

TerrainIrregularity

double ***Model_RuralITM::TerrainIrregularity***
Terrain irregularity describing the roughness of the terrain [m]

Mode

int ***Model_RuralITM::Mode***
Computation mode

- 0 = point to point mode

- 1 = area mode

ClimateCode

int ***Model_RuralITM::ClimateCode***
Climate code

- 0 = equatorial (Congo)
- 1 = continental subtropical (Sudan)
- 2 = maritime subtropical (West coast of Africa)
- 3 = desert (Sahara)
- 4 = continental temperate
- 5 = maritime temperate, over land (United Kingdom and continental west coasts)
- 6 = maritime temperate, over sea

CurvatureRadiusFactor

double ***Model_RuralITM::CurvatureRadiusFactor***
Factor for the radius of the earth

GroundPropertiesMode

int ***Model_RuralITM::GroundPropertiesMode***
Electrical properties of the ground

- 0 = Default ground properties as defined in Conductivity and Dielectricity
- 1 = Ground properties as defined in clutter classes

Conductivity

double ***Model_RuralITM::Conductivity***
Default conductivity of the ground [S/m] (relevant if GroundPropertiesMode is set to 0)

Dielectricity

double ***Model_RuralITM::Dielectricity***
Default dielectricity of the ground (relevant if GroundPropertiesMode is set to 0)

Reliability

double ***Model_RuralITM::Reliability***
Level of reliability. Reliability refers to a measure of the variability that a radio system observes during its use, allowed range [0.01 to 0.99]

Confidence

double ***Model_RuralITM::Confidence***
Level of confidence. Confidence refers to the variability that remains after specifying reliability, allowed range [0.01 to 0.99]

The documentation was generated from the following file:

- `source/Interface/WP_Model.h`

2.15.22 Model_RuralITU1546

Data Fields

float
Location

float
Time

int
RxHeightMode

Detailed Description

Parameter for the ITU-R P.1546 broadcasting model.

Remark Structure can be initialized with default values by using the function WinProp_Structure_Init_Model_RuralITU1546.

Field Documentation

Location

float ***Model_RuralITU1546::Location***
Location probability in percent, allowed range [0 to 100]

Time

float ***Model_RuralITU1546::Time***
Time variability in percent, allowed range [0 to 100]

RxHeightMode

int ***Model_RuralITU1546::RxHeightMode***
Receiver height mode in urban scenarios (clutters)

- 0 = normal height mode
- 1 = assume receiver height is always above urban reference height

The documentation was generated from the following file:

- `source/Interface/WP_Model.h`

2.15.23 Model_RuralITU526

Data Fields

int
ConsiderEarthCurvature

double
CurvatureRadiusFactor

Detailed Description

Parameter for the ITU-R P.526 model.

Remark Structure can be initialized with default values by using the function WinProp_Structure_Init_Model_RuralITU526.

Field Documentation

ConsiderEarthCurvature

int ***Model_RuralITU526::ConsiderEarthCurvature***
Parameter to consider earth curvature

- 0 = radius not considered
- 1 = radius considered

CurvatureRadiusFactor

double ***Model_RuralITU526::CurvatureRadiusFactor***
Factor for the radius of the earth

The documentation was generated from the following file:

- `source/Interface/WP_Model.h`

2.15.24 Model_RuralPE

Data Fields

int
GroundPropertiesMode

float
Conductivity

float
Dielectricity

int
PP_Mode

float
PP_MaxLossForward

float
PP_MaxLossReverse

float
PP_MaxLossDiff

int
CP_Mode

float
CP_ResolutionFactor

unsigned char
CP_UserDefResolutionFactor

float
CP_StretchFactor

unsigned char
CP_UserDefStretchFactor

int
SubModel

int
AtmosphereMode

float
MinDistance

unsigned char
UserDefMinDistance

int
FactorR

unsigned char
UserDefFactorR

float
MaxDeltaR

unsigned char
UserDefMaxDeltaR

float
FactorZ

unsigned char
UserDefFactorZ

Detailed Description

Parameter for the parabolic equation model.

Field Documentation

GroundPropertiesMode

int ***Model_RuralPE::GroundPropertiesMode***
Electrical properties of the ground

- 0 = Default ground properties as defined in Conductivity and Dielectricity

- 1 = Ground properties as defined in clutter classes

Conductivity

float ***Model_RuralPE::Conductivity***
Default conductivity of the ground (relevant if GroundPropertiesMode is set to 0)

Dielectricity

float ***Model_RuralPE::Dielectricity***
Default dielectricity of the ground (relevant if GroundPropertiesMode is set to 0)

PP_Mode

int ***Model_RuralPE::PP_Mode***
Post Processing mode

- 0 = Off
- 1 = Max increase or decrease of path loss per meter
- 2 = Upper boundary for path loss values

PP_MaxLossForward

float ***Model_RuralPE::PP_MaxLossForward***
Max increase of path loss in forward direction

PP_MaxLossReverse

float ***Model_RuralPE::PP_MaxLossReverse***
Max increase of path loss in reverse direction

PP_MaxLossDiff

float ***Model_RuralPE::PP_MaxLossDiff***
Max difference of path loss (adjacent pixels)

CP_Mode

int ***Model_RuralPE::CP_Mode***
Mode for vertical contour plot

- 0 = Off
- 1 = Without absorption media
- 2 = With absorption media

CP_ResolutionFactor

float ***Model_RuralPE::CP_ResolutionFactor***
Resolution factor

CP_UserDefResolutionFactor

unsigned char ***Model_RuralPE::CP_UserDefResolutionFactor***
Boolean to use a user-defined or an automatically determined value for the resolution factor

- true = Use the user-defined value CP_ResolutionFactor

- false = Use an automatically determined value

CP_StretchFactor

float ***Model_RuralPE::CP_StretchFactor***
Stretch factor for z-axis

CP_UserDefStretchFactor

unsigned char ***Model_RuralPE::CP_UserDefStretchFactor***
Boolean to use a user-defined or an automatically determined value for the stretch factor for z-axis

- true = Use the user-defined value CP_StretchFactor
- false = Use an automatically determined value

SubModel

int ***Model_RuralPE::SubModel***
Parabolic Equation Prediction (PE) Model

- 0 = Standard PE (discrete terrain approximation)
- 1 = Standard PE (continuous terrain approximation)
- 2 = Standard PE (terrain profile approximation with coordinate transformation)
- 3 = Wide Angle PE (discrete terrain approximation)
- 4 = Wide Angle PE (continuous terrain approximation)

AtmosphereMode

int ***Model_RuralPE::AtmosphereMode***
The atmosphere mode (altitude depending electrical properties)

- 0 = Constant dielectricity (Epsilon r = 1)
- 1 = Predefined height depending dielectricity

MinDistance

float ***Model_RuralPE::MinDistance***
Minimum distance to upper boundary

UserDefMinDistance

unsigned char ***Model_RuralPE::UserDefMinDistance***
Boolean to use a user-defined or an automatically determined value for the minimum distance to upper boundary

- true = Use the user-defined value MinDistance
- false = Use an automatically determined value

FactorR

int ***Model_RuralPE::FactorR***
Factor for increasing resolution of r (decreasing height)

UserDefFactorR

unsigned char ***Model_RuralPE::UserDefFactorR***
Boolean to use a user-defined or an automatically determined value for the factor for increasing resolution of r

- true = Use the user-defined value FactorR
- false = Use an automatically determined value

MaxDeltaR

float ***Model_RuralPE::MaxDeltaR***
Maximum distance between two succeeding points

UserDefMaxDeltaR

unsigned char ***Model_RuralPE::UserDefMaxDeltaR***
Boolean to use a user-defined or an automatically determined value for the maximum distance between two succeeding points

- true = Use the user-defined value MaxDeltaR
- false = Use an automatically determined value

FactorZ

float ***Model_RuralPE::FactorZ***
Factor for vertical resolution of computation grid

UserDefFactorZ

unsigned char ***Model_RuralPE::UserDefFactorZ***
Boolean to use a user-defined or an automatically determined value for the factor for vertical resolution

- true = Use the user-defined value FactorZ
- false = Use an automatically determined value

The documentation was generated from the following file:

- `source/Interface/WP_Model.h`

2.15.25 Model_UrbanIRT

Data Fields

INTERFACE_MATERIAL
DefaultMaterial

INTERFACE_MATERIAL
GroundMaterial

double
BreakpointExponentBefore

double
BreakpointExponentAfter

double
BreakpointExponentBeforeLOS

double
BreakpointExponentAfterLOS

double
BreakpointExponentSat

unsigned char
PostProcessingMode

unsigned char
Postprocessing

char
DiffractionModel

char
MaxReflections

char
MaxDiffractions

char
MaxScatterings

char
MaxSumReflDiff

unsigned char
GroundReflection

double
AngularTolerance

unsigned char
RaysPostProcessSRT

short
KE_Interactions

double
KE_DiffractionLossOffset

double
KE_DiffractionLossAngle

int
KE_IRT_Interactions

unsigned char
FreeSpaceCancel

unsigned char
Superposition

unsigned char
COST_Transition

double
COST_Trans_Min

double
COST_Trans_Max

int
COST_Trans_Add_Ctrl

double
COST_Trans_Add_Min_Dist

double
COST_Trans_Add_Max_Dist

int
COST_Trans_Add_Min_Build

int
COST_Trans_Add_Max_Build

double
COST_TransDistance

Detailed Description

Model_UrbanIRT: Parameter for urban IRT model. See

- WinProp_ParaMain
- OutdoorPlugIn_ComputePrediction

This structure is used for the definition of the parameters for the Intelligent Ray Tracing Model. The structure can be passed to the OutdoorPlugIn_ComputePrediction function but this is optional. It can be initialized with default values by using WinProp_Structure_Init_Model_UrbanIRT(). Determination of Diffraction Loss using the Empirical Interaction Model If the empirical interaction model is turned on (DiffractionModel = 'e') for the determination of the diffraction loss the following parameters are considered:

- LossDiffractionMin (LInc,min)
- LossDiffractionMax (LInc,max)
- LossDiffractionOut (LDiff)

For the empirical diffraction model the total loss of the diffracted rays is computed depending on the angles Phi and Phi' using the following relations indicated in the next figures:

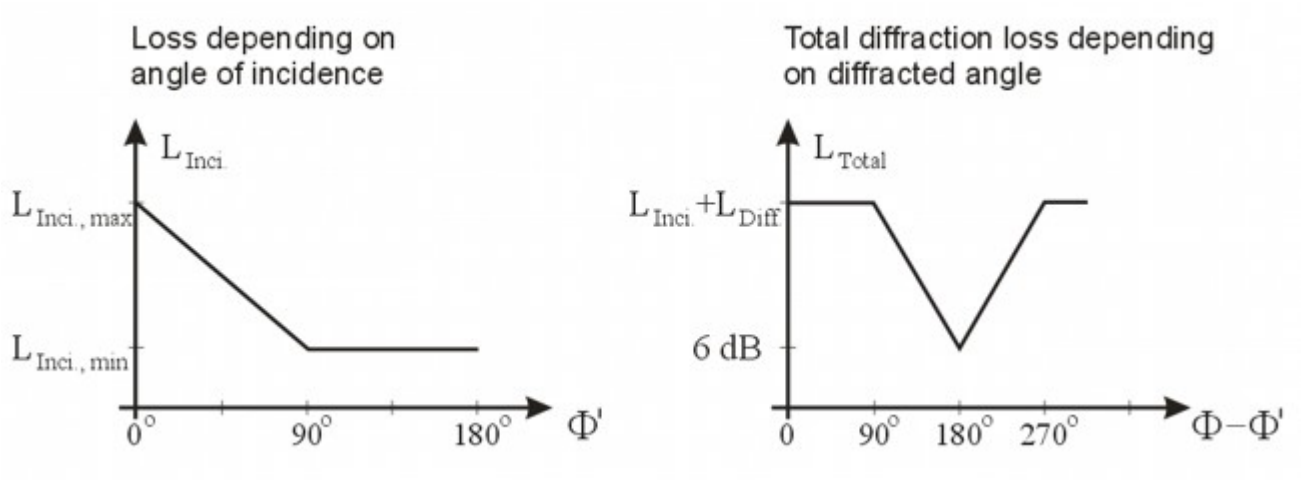

Figure 5: Empirical Diffraction Model

The empirical diffraction model computes the total diffraction loss in a two step approach based on the three parameters LInc,min, LInc,max and LDiff. In the first step the loss depending on the angle of incidence is determined (see left figure above). For this purpose the first two parameters LInc,min and LInc,max are evaluated. The corresponding loss increases with decreasing angle of incidence (i.e. increasing grazing incidence). Based on this first result the second curve (shown in the right figure above) is evaluated, leading to the total diffraction loss. Basically the diffraction loss increases with increasing interaction angle. However for a difference of 180 degrees, the total diffraction loss is fixed to 6 dB as in this special case the incident wave propagates straight forward while half of the space is shadowed according to the given obstacle. Therefore the range of possible total diffraction losses is given by [6 dB; LInc,max + LDiff]. The given angle dependence is derived from the uniform diffraction theory (UTD) by the evaluation of measurements with different materials (brick, concrete) in an anechoic chamber and can be varied with the parameters LInc,min, LInc,max and LDiff within appropriate limits.

Field Documentation

DefaultMaterial

INTERFACE_MATERIAL ***Model_UrbanIRT::DefaultMaterial***

GroundMaterial

INTERFACE_MATERIAL ***Model_UrbanIRT::GroundMaterial***

BreakpointExponentBefore

double ***Model_UrbanIRT::BreakpointExponentBefore***
Path Loss Exponent before breakpoint for NLOS areas.

BreakpointExponentAfter

double ***Model_UrbanIRT::BreakpointExponentAfter***
Path Loss Exponent after breakpoint for NLOS areas.

BreakpointExponentBeforeLOS

double ***Model_UrbanIRT::BreakpointExponentBeforeLOS***
Path Loss Exponent before breakpoint for LOS areas.

BreakpointExponentAfterLOS

double ***Model_UrbanIRT::BreakpointExponentAfterLOS***
Path Loss Exponent after breakpoint for LOS areas.

BreakpointExponentSat

double ***Model_UrbanIRT::BreakpointExponentSat***
Path Loss Exponent for a satellite transmitter.

PostProcessingMode

unsigned char ***Model_UrbanIRT::PostProcessingMode***
Post-processing mode

- 0 = none
- 2 = KE (Knife edge)

Postprocessing

unsigned char ***Model_UrbanIRT::Postprocessing***
Boolean to enable/disable post-processing with Knife edge model

- true = enabled
- false = disabled

DiffractionModel

char ***Model_UrbanIRT::DiffractionModel***
Interaction model for IRT prediction model

- 'g' = fresnel & GTD/UTD
- 'e' = empirical interaction model

MaxReflections

char ***Model_UrbanIRT::MaxReflections***
Maximum number of reflections

MaxDiffractions

char ***Model_UrbanIRT::MaxDiffractions***
Maximum number of diffractions

MaxScatterings

char ***Model_UrbanIRT::MaxScatterings***
Maximum number of scatterings

MaxSumReflDiff

char ***Model_UrbanIRT::MaxSumReflDiff***
Maximum number of reflections + diffractions + scatterings.

GroundReflection

unsigned char ***Model_UrbanIRT::GroundReflection***
Boolean to enable/disable ground reflection

- true = enabled
- false = disabled

AngularTolerance

double ***Model_UrbanIRT::AngularTolerance***
Max. angular tolerance for ground reflection.

RaysPostProcessSRT

unsigned char ***Model_UrbanIRT::RaysPostProcessSRT***
Boolean to enable/disable postprocessing of IRT rays with SRT

- true = enabled
- false = disabled

KE_Interactions

short ***Model_UrbanIRT::KE_Interactions***
Max. number of interactions for Knife-Edge Model (if postprocessing with Knife-Edge Model is enabled).

KE_DiffractionLossOffset

double ***Model_UrbanIRT::KE_DiffractionLossOffset***
Offset (dB) for Knife-Edge diffraction computation (if postprocessing with Knife-Edge Model is enabled).

KE_DiffractionLossAngle

double ***Model_UrbanIRT::KE_DiffractionLossAngle***
Max. diffraction loss (dB) for Knife-Edge Model (if postprocessing with Knife-Edge Model is enabled).

KE_IRT_Interactions

int ***Model_UrbanIRT::KE_IRT_Interactions***
1 to activate the combined KE & IRT interactions, 0 otherwise

FreeSpaceCancel

unsigned char ***Model_UrbanIRT::FreeSpaceCancel***
Boolean to enable/disable abortion of computation of rays for one receiver pixel if free space loss is reached.

- true = enabled

- false = disabled

Superposition

unsigned char ***Model_UrbanIRT::Superposition***
Mode for superposition of rays

- 0 = Uncorrelated
- 1 = Coherent (with phase)

COST_Transition

unsigned char ***Model_UrbanIRT::COST_Transition***
Parameter for COST postprocessing. Please leave the default value.

COST_Trans_Min

double ***Model_UrbanIRT::COST_Trans_Min***
Parameter for COST postprocessing. Please leave the default value.

COST_Trans_Max

double ***Model_UrbanIRT::COST_Trans_Max***
Parameter for COST postprocessing. Please leave the default value.

COST_Trans_Add_Ctrl

int ***Model_UrbanIRT::COST_Trans_Add_Ctrl***
Parameter for COST postprocessing. Please leave the default value.

COST_Trans_Add_Min_Dist

double ***Model_UrbanIRT::COST_Trans_Add_Min_Dist***
Parameter for COST postprocessing. Please leave the default value.

COST_Trans_Add_Max_Dist

double ***Model_UrbanIRT::COST_Trans_Add_Max_Dist***
Parameter for COST postprocessing. Please leave the default value.

COST_Trans_Add_Min_Build

int ***Model_UrbanIRT::COST_Trans_Add_Min_Build***
Parameter for COST postprocessing. Please leave the default value.

COST_Trans_Add_Max_Build

int ***Model_UrbanIRT::COST_Trans_Add_Max_Build***
Parameter for COST postprocessing. Please leave the default value.

COST_TransDistance

double ***Model_UrbanIRT::COST_TransDistance***
Parameter for COST postprocessing. Please leave the default value.

The documentation was generated from the following file:

- source/Interface/WP_Model.h

2.15.26 Model_UrbanITU1411

Data Fields

int
AddGaussian

int
UseStatisticlModel

float
LocationPercent

int
PropEnvironment

int
PropModel

int
PropSituation

int
LosMode

int
TrafficMode

int
IgnoreBreakPoint

float
AdditionalLoss

int
UseSiteGeneral

Detailed Description

This structure is used to set the parameters of ITU-R P.1411 model.

Remark Structure can be initialized with default values by using the function WinProp_Structure_Init_Model_UrbanITU1411.

Field Documentation

AddGaussian

int ***Model_UrbanITU1411::AddGaussian***
Add Gaussian random distribution with a specific standard deviation (according to the propagation environment and to the propagation situation) to the predicted loss. Only relevant for site-general models.

- 0 = disabled
- 1 = enabled

UseStatisticlModel

int ***Model_UrbanITU1411::UseStatisticlModel***
Use a statistical model to determine LOS/NLOS region Determination mode of LOS/NLOS regions instead. Only relevant for site-general models

- 0 = disabled (the building data will be used to determine LOS/NLOS regions)
- 1 = enabled (the statistical model will be used to determine LOS/NLOS regions)

LocationPercent

float ***Model_UrbanITU1411::LocationPercent***
Location percentage for the statistical model. Only relevant for site-general models that use the statistical model to determine LOS/NLOS regions.

PropEnvironment

int ***Model_UrbanITU1411::PropEnvironment***
Propagation environment

- 0 = Urban very high rise
- 1 = Urban / Urban high rise
- 2 = SubUrban / Urban low rise
- 3 = Residential

PropModel

int ***Model_UrbanITU1411::PropModel***
Propagation models

- 0 = Site-specific
- 1 = Site-general

PropSituation

int ***Model_UrbanITU1411::PropSituation***
Propagation situation in Non-Line-of-Sight (NLOS) configurations

- 0 = Propagation over roof-tops
- 1 = Propagation within street canyons
- 2 = Propagation near street level
- 3 = Auto detection

LosMode

int ***Model_UrbanITU1411::LosMode***
Calculation mode in Line-of-Sight (LOS) configurations. Only relevant for site-specific models.

- 0 = Lower bound
- 1 = Upper bound

- 2 = Median value

TrafficMode

int ***Model_UrbanITU1411::TrafficMode***
Traffic mode (affects the break-point distance). Only relevant if break-point distance is considered.

- 0 = Light
- 1 = Heavy

IgnoreBreakPoint

int ***Model_UrbanITU1411::IgnoreBreakPoint***
Ignore break-point distance

- 0 = disabled (the break-point distance is considered)
- 1 = enabled (the break-point distance is ignored)

AdditionalLoss

float ***Model_UrbanITU1411::AdditionalLoss***
Additional loss in dB. Only applied if the transmitter is located inside a building.

UseSiteGeneral

int ***Model_UrbanITU1411::UseSiteGeneral***
Use site general models to complete site specific models if the site-specific gives unpredicted pixel. Only relevant for site-specific models.

- 0 = disabled
- 1 = enabled

The documentation was generated from the following file:

- `source/Interface/WP_Model.h`

2.15.27 Model_VRT

Data Fields

int
DirectRay

int
SingleReflection

int
SingleDiffraction

int
ReflectionDiffraction

int
DiffractionReflection

int
DoubleDiffraction

int
KnifeEdgeDiffraction

int
OnlyForwardPropagation

int
TopoClutterInteractions

int
VectorInteractions

int
ScatteringInteractions

int
ConsiderPhase

double
MinAngleDiffraction

double
ExtensionBehindReceiver

double
AngleToleranceReflection

VRT_KE_PARA
KnifeEdgePara

Detailed Description

This structure is used to set the model parameters of the VRT (Vertical Ray Tracing). This structure can be passed optionally to the WinProp_Predict or WinProp_Predict_Points functions.

Remark Structure can be initialized with default values by using the function WinProp_Structure_Init_Model_VRT.

Field Documentation

DirectRay

int ***Model_VRT::DirectRay***
Compute direct ray.

SingleReflection

int ***Model_VRT::SingleReflection***
Compute rays with single reflections.

SingleDiffraction

int ***Model_VRT::SingleDiffraction***
Compute rays with single diffractions.

ReflectionDiffraction

int ***Model_VRT::ReflectionDiffraction***
Compute rays with one reflection and one diffraction.

DiffractionReflection

int ***Model_VRT::DiffractionReflection***
Compute rays with one diffraction and one reflection.

DoubleDiffraction

int ***Model_VRT::DoubleDiffraction***
Compute rays with two diffractions.

KnifeEdgeDiffraction

int ***Model_VRT::KnifeEdgeDiffraction***
Compute rays with multiple diffractions (knife-edge model).

OnlyForwardPropagation

int ***Model_VRT::OnlyForwardPropagation***
Compute rays only in forward direction.

TopoClutterInteractions

int ***Model_VRT::TopoClutterInteractions***
Compute interactions at topography and clutter/land usage.

VectorInteractions

int ***Model_VRT::VectorInteractions***
Compute interactions at vector objects.

ScatteringInteractions

int ***Model_VRT::ScatteringInteractions***
Compute 3D Scattering at Topography/Clutter objects.

ConsiderPhase

int ***Model_VRT::ConsiderPhase***
Consider phase for superposition of rays (coherent superposition).

MinAngleDiffraction

double ***Model_VRT::MinAngleDiffraction***
Minimum angle between two polygons for consideration of diffraction.

ExtensionBehindReceiver

double ***Model_VRT::ExtensionBehindReceiver***
Additional distance behind receiver for determination of vertical plane.

AngleToleranceReflection

double ***Model_VRT::AngleToleranceReflection***
Angular tolerance for ground reflection in deg.

KnifeEdgePara

VRT_KE_PARA ***Model_VRT::KnifeEdgePara***
Parameters of the multiple knife-edge diffraction model.

The documentation was generated from the following file:

- `source/Interface/WP_Model.h`

2.15.28 PLANEPOINT

Data Fields

COORDPOINT
Location

double
Value

int
Computed

Detailed Description

This structure describes the result for one pixel. It is included in the structure WinProp_Result_Plane.

Field Documentation

Location

COORDPOINT ***PLANEPOINT::Location***
Location of point (X, Y, Z in meter).

Value

double ***PLANEPOINT::Value***
Computed result (e.g. field strength, path loss, power, SNIR, best server). Depends on the operation made.

Computed

int ***PLANEPOINT::Computed***
Indicates if a pixel was computed (1) or not (0).

The documentation was generated from the following file:

- `source/Interface/WP_Result.h`

2.15.29 RCS_PARA

Data Fields

int
GroupId

WinProp_Filename
Filename

COORDPOINT
Position

float
Orientation

float
AngularFilter

int
Id

Detailed Description

This structure is used to set the RCS parameters which can by used with the Standard Ray Tracing model (SRT). It can be initialized with initial values by using WinProp_Structure_Init_ParameterRCS.

Field Documentation

GroupId

int ***RCS_PARA::GroupId***
Group Id of the group that contains the RCS.

Filename

WinProp_Filename ***RCS_PARA::Filename***
Filename of the .ffe RCS file.

Position

COORDPOINT ***RCS_PARA::Position***
RCS position (shift) with respect to the group center.

Orientation

float ***RCS_PARA::Orientation***
Azimuth orientation of the RCS in degrees. East (0 deg) to north (90 deg).

AngularFilter

float ***RCS_PARA::AngularFilter***
Angular averaging filter in degrees.

Id

int ***RCS_PARA::Id***
A unique ID for the RCS.

The documentation was generated from the following file:

- `source/Interface/WP_Model.h`

2.15.30 TOPOGRAPHY

Data Fields

double
resolutionX

double
resolutionY

double
lowerLeftX

double
lowerLeftY

double
upperRightX

double
upperRightY

unsigned int
columns

unsigned int
rows

int
coordDatum

int
coordEllipsoid

int
coordGeoUTM

int
coordUTMZone

bool
coordUTMNorth

float **
heights

Detailed Description

Topography definition.

Field Documentation

resolutionX

double ***TOPOGRAPHY::resolutionX***
resolution in x-coordinate

resolutionY

double ***TOPOGRAPHY::resolutionY***
resolution in y-coordinate

lowerLeftX

double ***TOPOGRAPHY::lowerLeftX***
Topology boundaries on a Cartesian plane.

lowerLeftY

double ***TOPOGRAPHY::lowerLeftY***
Topology boundaries on a Cartesian plane.

upperRightX

double ***TOPOGRAPHY::upperRightX***
Topology boundaries on a Cartesian plane.

upperRightY

double ***TOPOGRAPHY::upperRightY***
Topology boundaries on a Cartesian plane.

columns

unsigned int ***TOPOGRAPHY::columns***
Number of columns.

rows

unsigned int ***TOPOGRAPHY::rows***
Number of rows.

coordDatum

int ***TOPOGRAPHY::coordDatum***
datum of ccordinate system => Datum.h

coordEllipsoid

int ***TOPOGRAPHY::coordEllipsoid***
ellipsoid of ccordinate system => CoordinateSystem.h

coordGeoUTM

int ***TOPOGRAPHY::coordGeoUTM***
UTM Coords (1) or Geodetic Coords (0)

coordUTMZone

int ***TOPOGRAPHY::coordUTMZone***
UTM Zone if UTM Coords

coordUTMNorth

bool ***TOPOGRAPHY::coordUTMNorth***
Hemisphere if UTM Coords (North=true, South=false)

heights

float ** ***TOPOGRAPHY::heights***
matrix of heights of each point in the topography

The documentation was generated from the following file:

- `source/Public/Interface/InterfaceType.h`

2.15.31 URBAN_BUILDING

Data Fields

int
ID

int
buildingType

int
nbrCorners

INTERFACE_CORNER *
corners

INTERFACE_MATERIAL *
material

URBAN_BUILDING *
next

Detailed Description

Urban building definition.

Field Documentation

ID

int ***URBAN_BUILDING::ID***
unique ID of building

buildingType

int ***URBAN_BUILDING::buildingType***
type of building

nbrCorners

int ***URBAN_BUILDING::nbrCorners***
number of corners

corners

INTERFACE_CORNER * ***URBAN_BUILDING::corners***
corners of building

material

INTERFACE_MATERIAL * ***URBAN_BUILDING::material***
material parameters of building

next

URBAN_BUILDING * ***URBAN_BUILDING::next***
pointer to next building

The documentation was generated from the following file:

- `source/Public/Interface/InterfaceType.h`

2.15.32 URBAN_BUILDINGS

Data Fields

double
lowerLeftX

double
lowerLeftY

double
upperRightX

double
upperRightY

double
minBuildingHeight

double
parameter

int
nbrBuildings

int
maxNbrCorners

URBAN_BUILDING *
buildingData

Detailed Description

Urban buildings in database.

Field Documentation

lowerLeftX

double ***URBAN_BUILDINGS::lowerLeftX***
lower left coords of prediction area

lowerLeftY

double ***URBAN_BUILDINGS::lowerLeftY***
lower right coords of prediction area

upperRightX

double ***URBAN_BUILDINGS::upperRightX***
upper right corner of prediction area

upperRightY

double ***URBAN_BUILDINGS::upperRightY***
upper left corner of prediction area

minBuildingHeight

double ***URBAN_BUILDINGS::minBuildingHeight***
minimum height for buildings to be considered

parameter

double ***URBAN_BUILDINGS::parameter***
building type specific parameter

nbrBuildings

int ***URBAN_BUILDINGS::nbrBuildings***
total number of buildings

maxNbrCorners

int ***URBAN_BUILDINGS::maxNbrCorners***
maximum number of corners of a building

buildingData

URBAN_BUILDING * ***URBAN_BUILDINGS::buildingData***
urban building definition

The documentation was generated from the following file:

- `source/Public/Interface/InterfaceType.h`

2.15.33 VRT_KE_PARA

Data Fields

int
SubModel

int
LimitedNrDiffractions

int
MaxNrDiffractions

float
PathLossExponentBefore

float
PathLossExponentAfter

int
BreakpointMode

float
BreakpointFactor

float
BreakpointOffset

float
BreakpointDistance

Detailed Description

This structure is used to set the parameters of the knife edge diffraction used in the VRT (Vertical Ray Tracing). This structure is used in Model_VRT.

Remark Structure can be initialized with default values by using the function WinProp_Structure_Init_Model_VRT.

Field Documentation

SubModel

int ***VRT_KE_PARA::SubModel***
Mode for Knife-Edge computation method

- 1 = Deygout
- 2 = Epstein-Petersen

LimitedNrDiffractions

int ***VRT_KE_PARA::LimitedNrDiffractions***
Mode for Knife-Edge number of diffractions

- 0 = Unlimited number of diffraction
- 1 = Limited number of diffractions determined by MaxNrDiffractions

MaxNrDiffractions

int ***VRT_KE_PARA::MaxNrDiffractions***
Maximum number of allowed diffractions.

PathLossExponentBefore

float ***VRT_KE_PARA::PathLossExponentBefore***
Path loss exponent before breakpoint.

PathLossExponentAfter

float ***VRT_KE_PARA::PathLossExponentAfter***
Path loss exponent before breakpoint.

BreakpointMode

int ***VRT_KE_PARA::BreakpointMode***
Mode for breakpoint calculation

- 0 = Automatic mode
- 1 = User-defined determined by BreakpointDistance

BreakpointFactor

float ***VRT_KE_PARA::BreakpointFactor***
Factor used for determining the breakpoint distance in the automatic mode (default = 4 * PI).

BreakpointOffset

float ***VRT_KE_PARA::BreakpointOffset***
Offset distance to be added to the automatically determined distance.

BreakpointDistance

float ***VRT_KE_PARA::BreakpointDistance***
User-defined breakpoint distance.

The documentation was generated from the following file:

- `source/Interface/WP_Model.h`

2.15.34 WinProp_Additional

Data Fields

WinProp_Propagation_Results *
OutputResults

int
FullPolarimetric

int
MultipleWallCorrection

double
MultipleWallDistance

int
MaxPathLossEnabled

float
MaxPathLoss

int
MaxDynamicPathsEnabled

float
MaxDynamicPaths

int
MaxNumberPathsUsed

int
MaxNumberPaths

int
RaysLogLin

int
RaysPowerField

int
ResultFiltering

int
ResultFilterOrder

double
ResultOffset

int
UseDynamicAntennaRange

double
DynamicAntennaRange

int
PredictionPlanes

int
PredictionSurfaces

*double **
TimeInstances

int
NbrTimeInstances

bool
IndividualMaterials

WINPROP_INTERACTION_MODEL
InteractionModel

int
MultiThreading

WINPROP_BREAKPOINT_MODE
BreakpointMode

double
BreakpointFactor

double
BreakpointOffset

double
BreakpointDistance

WINPROP_RAD_CABLE_MODEL
RadCableModelID

int
RadCableTransAngleDependency

double
RadCableExpLOS

double
RadCableExpNLOS

double
RadCableStepDistance

WinProp_Additional_Freq_Loss
FreqDepLosses

INTERFACE_MATERIAL
DefaultMaterial

INTERFACE_MATERIAL
DefaultGroundMaterial

int
ConsiderDatabasePolygons

int
NrDatabasePolygons

WinProp_Polygon *
DatabasePolygons

int
NrWinPropGroupDynamics

WinProp_Group_Dynamics *
WinPropGroupDynamicSets

Detailed Description

This structure is used to pass additional settings to the WinProp_Predict function. Passing the structure is optional.

Field Documentation

OutputResults

WinProp_Propagation_Results * ***WinProp_Additional::OutputResults***
This structure allows to define additional results to be computed and written in binary WinProp format. Disabled by setting to NULL.

FullPolarimetric

int ***WinProp_Additional::FullPolarimetric***
Full polarimetric simulation (enabled/disabled)

- 0 = standard polarimetric analysis with total gain pattern only
- 1 = full polarimetric analysis with .ffe file or two gains patterns, one for vertical and one for horizontal polarisation

MultipleWallCorrection

int ***WinProp_Additional::MultipleWallCorrection***
Correction of multiple defined walls?

MultipleWallDistance

double ***WinProp_Additional::MultipleWallDistance***
Distance value for multiple walls featureSee also MultipleWallCorrection

MaxPathLossEnabled

int ***WinProp_Additional::MaxPathLossEnabled***
Max absolute value of path loss of rays used to compute prediction results (enabled/disabled)

- 0 = don't limit max absolute value of path loss per ray
- 1 = limit max absolute value of path loss per ray, maximum has to be specified in MaxPathLoss

MaxPathLoss

float ***WinProp_Additional::MaxPathLoss***
The max absolute value of path loss of rays. Only applicable if MaxPathLossEnabled is set to 1

MaxDynamicPathsEnabled

int ***WinProp_Additional::MaxDynamicPathsEnabled***
Max dynamic range of path loss of rays used to compute prediction results (enabled/ disabled)

- 0 = don't limit the dynamic range of rays
- 1 = limit the dynamic range of rays, maximum has to be specified in MaxDynamicPaths

MaxDynamicPaths

float ***WinProp_Additional::MaxDynamicPaths***
The maximum dynamic range path loss of rays. Only applicable if MaxDynamicPathsEnabled is set to 1

MaxNumberPathsUsed

int ***WinProp_Additional::MaxNumberPathsUsed***
Limit the number of rays which are computed.

- 0 = don't limit the number of rays
- 1 = limit the number of rays, maximum has to be specified in MaxNumberPaths

MaxNumberPaths

int ***WinProp_Additional::MaxNumberPaths***
The maximum number rays. Only applicable if MaxNumberPathsUsed is set to 1

RaysLogLin

int ***WinProp_Additional::RaysLogLin***
Define if the delay spread and angular spreads are computed

- 0 = with logarithmic values
- 1 = with linear values

RaysPowerField

int ***WinProp_Additional::RaysPowerField***
Define if the delay spread and angular spreads are computed based on

- 0 = power values
- 1 = field strength values

ResultFiltering

int ***WinProp_Additional::ResultFiltering***
Apply filtering after prediction? (1) - Yes, (0) - No.

ResultFilterOrder

int ***WinProp_Additional::ResultFilterOrder***
The result filter orderSee also ResultFiltering

ResultOffset

double ***WinProp_Additional::ResultOffset***
Offset to be added to all pixels

UseDynamicAntennaRange

int ***WinProp_Additional::UseDynamicAntennaRange***
Set to 1 to limit the dynamic range of antenna patterns

DynamicAntennaRange

double ***WinProp_Additional::DynamicAntennaRange***
The max dynamic range of antenna patternsSee also UseDynamicAntennaRange

PredictionPlanes

int ***WinProp_Additional::PredictionPlanes***
Enable prediction on planes defined in the database

- 0 = Disable prediction on planes
- 1 = Enable prediction on planes

PredictionSurfaces

int ***WinProp_Additional::PredictionSurfaces***
Enable prediction on surfaces defined in the database

- 0 = Disable prediction on surfaces
- 1 = Enable prediction on surfaces

TimeInstances

*double ** ***WinProp_Additional::TimeInstances***
Time instances for time-variant simulation

NbrTimeInstances

int ***WinProp_Additional::NbrTimeInstances***
Number of time instances

IndividualMaterials

bool ***WinProp_Additional::IndividualMaterials***
Individual materials or not. Only applicable in combination with WinProp database in WinProp_Scenario::VectorDatabase

- false = default materials
- true = individual materials

InteractionModel

WINPROP_INTERACTION_MODEL ***WinProp_Additional::InteractionModel***
Type of interaction model.

MultiThreading

int ***WinProp_Additional::MultiThreading***
Number of threads being used for the computation of the particular antenna. Needs to be greater than zero. Currently only supported by the Standard Ray Tracing (SRT).

BreakpointMode

WINPROP_BREAKPOINT_MODE ***WinProp_Additional::BreakpointMode***
Computation of breakpoint distance.

BreakpointFactor

double ***WinProp_Additional::BreakpointFactor***
Factor for determination of breakpoint distance. Only required if BreakpointMode is set to BREAKPOINT_DEFAULT.

BreakpointOffset

double ***WinProp_Additional::BreakpointOffset***
Offset (meter) for determination of breakpoint distance. Only required if BreakpointMode is set to BREAKPOINT_DEFAULT.

BreakpointDistance

double ***WinProp_Additional::BreakpointDistance***
Fixed distance (meter) for breakpoint. Only required if BreakpointMode is set to BREAKPOINT_FIXED.

RadCableModelID

WINPROP_RAD_CABLE_MODEL ***WinProp_Additional::RadCableModelID***
Propagation model used for radiating cables.

RadCableTransAngleDependency

int ***WinProp_Additional::RadCableTransAngleDependency***
Angle dependency for transmission loss (enabled/disabled)

- 0 = don't consider angle of incident for transmission loss
- 1 = consider angle of incident for transmission loss

RadCableExpLOS

double ***WinProp_Additional::RadCableExpLOS***
Path loss exponent for LOS areas for radiating cables.

RadCableExpNLOS

double ***WinProp_Additional::RadCableExpNLOS***
Path loss exponent for NLOS areas for radiating cables.

RadCableStepDistance

double ***WinProp_Additional::RadCableStepDistance***
Discretization distance of radiating cable.

FreqDepLosses

WinProp_Additional_Freq_Loss ***WinProp_Additional::FreqDepLosses***
This structure allows to define additional frequency dependent losses (atmospheric losses).

DefaultMaterial

INTERFACE_MATERIAL ***WinProp_Additional::DefaultMaterial***
Default material properties for walls. Only required if IndividualMaterials is set to false.

DefaultGroundMaterial

INTERFACE_MATERIAL ***WinProp_Additional::DefaultGroundMaterial***
Default material properties for the ground. Only required if IndividualMaterials is set to false

ConsiderDatabasePolygons

int ***WinProp_Additional::ConsiderDatabasePolygons***
Consider the database polygons if they are defined in the scenario (enabled/disabled)

- 0 = don't consider database polygons
- 1 = consider database polygons

NrDatabasePolygons

int ***WinProp_Additional::NrDatabasePolygons***
Number of database polygons in the scenario.

DatabasePolygons

WinProp_Polygon * ***WinProp_Additional::DatabasePolygons***
The database polygons in the scenario.

NrWinPropGroupDynamics

int ***WinProp_Additional::NrWinPropGroupDynamics***
Number of group dynamics.

WinPropGroupDynamicSets

WinProp_Group_Dynamics * ***WinProp_Additional::WinPropGroupDynamicSets***
Dynamic sets of moving groups

The documentation was generated from the following file:

- `source/Interface/WP_Propagation.h`

2.15.35 WinProp_Additional_Freq_Loss

Data Fields

int
Enabled

int
NrValues

*float **
FrequencyMHz

*float **
AddLossPerMeter

Detailed Description

This structure is used to define additional frequency-dependent losses (atmospheric losses). In this structure, the frequencies are defined as single values, then the additional losses for any frequency will be determined internally. If the transmitter frequency is lower than or equal to the lowest defined frequency, the losses of the lowest defined frequency will be considered. If the transmitter frequency is higher than or equal to the highest defined frequency, the losses of the highest frequency will be considered. If the transmitter frequency is between two defined frequencies, an interpolated value between the losses of the corresponding frequencies will be considered.

Field Documentation

Enabled

int ***WinProp_Additional_Freq_Loss::Enabled***
Additional frequency dependent losses:

- 0: disabled
- 1: enabled

NrValues

int ***WinProp_Additional_Freq_Loss::NrValues***
Number of frequency-depended losses.

FrequencyMHz

*float ** ***WinProp_Additional_Freq_Loss::FrequencyMHz***
Array of frequencies in MHz. Array of length NrValues.

AddLossPerMeter

*float ** ***WinProp_Additional_Freq_Loss::AddLossPerMeter***
Array of additional losses per meter (for each frequency). Array of length NrValues.

The documentation was generated from the following file:

- `source/Interface/WP_Propagation.h`

2.15.36 WinProp_Additional_Internal

Data Fields

int
TotalPower

TOTAL_POWER_MODE
TotalPowerMode

int
BestServer

int
BestServerASCII

int
MaxThroughput

int
MaxThroughputASCII

int
StreamsMIMO

int
EMC

WinProp_Filename
EMC_Database

int
ReceivedPower

int
SNIR

int
HandoverType

int
NrReceivedCarriers

int
NrReceivedTRXs

int
MinNumberTRX

int
NrReceivedSites

int
MinNumberSites

int
NeighborCells

int
NeighborCellOrder

int
RecPowerInterNoise

int
RecInterNoise

int
ReferenceSignals

int
TransTxPower

int
TransRecPower

int
TransRecProbability

int
TransSNIR

int
TransNrParallelStreams

int
TransDataRates

int
TrafficState

int
TrafficOffered

int
TrafficServed

int
TrafficBlocked

int
NrUsersGenerated

WinProp_Filename
AirInterfaceFile

Detailed Description

This structure is used to pass internal additional settings to the WinProp_ProjectIndoorWrite function.

Field Documentation

TotalPower

int ***WinProp_Additional_Internal::TotalPower***
Total power superposition

- 0 = disabled
- 1 = enabled

TotalPowerMode

TOTAL_POWER_MODE ***WinProp_Additional_Internal::TotalPowerMode***
Total power calculation mode

- TOTAL_POWER_MODE::INCOHERENT = Incoherent superposition
- TOTAL_POWER_MODE::COHERENT = Coherent superposition

BestServer

int ***WinProp_Additional_Internal::BestServer***
Best server (Cell assignment)

- 0 = disabled
- 1 = enabled

BestServerASCII

int ***WinProp_Additional_Internal::BestServerASCII***
Additional ASCII output of best server

- 0 = no additional output
- 1 = additional output of cell assignment (reports only received cells)
- 2 = additional output of cell assignment (reports all cells)

MaxThroughput

int ***WinProp_Additional_Internal::MaxThroughput***
Maximum achievable throughput

- 0 = disabled
- 1 = enabled

MaxThroughputASCII

int ***WinProp_Additional_Internal::MaxThroughputASCII***
Additional ASCII output of maximum achievable throughput 0 = no additional output 1 = additional output of data streams

StreamsMIMO

int ***WinProp_Additional_Internal::StreamsMIMO***
Number of received MIMO streams

- 0 = disabled

- 1 = enabled

EMC

int ***WinProp_Additional_Internal::EMC***
EMC Analysis

- 0 = disabled
- 1 = enabled

EMC_Database

WinProp_Filename ***WinProp_Additional_Internal::EMC_Database***
EMC specification filename

ReceivedPower

int ***WinProp_Additional_Internal::ReceivedPower***
Serving Carrier: Received Power (Cell Assignment)

- 0 = disabled
- 1 = enabled

SNIR

int ***WinProp_Additional_Internal::SNIR***
Serving Carrier: SNIR (Cell Assignment)

- 0 = disabled
- 1 = enabled

HandoverType

int ***WinProp_Additional_Internal::HandoverType***
Type of the handover in CDMA

- 0 = disabled
- 1 = enabled

NrReceivedCarriers

int ***WinProp_Additional_Internal::NrReceivedCarriers***
Number of all carriers received by MS

- 0 = disabled
- 1 = enabled

NrReceivedTRXs

int ***WinProp_Additional_Internal::NrReceivedTRXs***
Number of all TRXs received by MS

- 0 = disabled
- 1 = enabled

MinNumberTRX

int ***WinProp_Additional_Internal::MinNumberTRX***
Minimum required number of TRXs (if less received, no output for pixel). Relevant if NrReceivedTRXs = 1

NrReceivedSites

int ***WinProp_Additional_Internal::NrReceivedSites***
Number of all sites received by MS

- 0 = disabled
- 1 = enabled

MinNumberSites

int ***WinProp_Additional_Internal::MinNumberSites***
Minimum required number of sites (if less received, no output for pixel). Relevant if NrReceivedSites = 1

NeighborCells

int ***WinProp_Additional_Internal::NeighborCells***
Neighbor cell list

- 0 = disabled
- 1 = enabled

NeighborCellOrder

int ***WinProp_Additional_Internal::NeighborCellOrder***
Maximum distance between pixels to be considered as neighbors. Relevant if NeighborCells = 1

RecPowerInterNoise

int ***WinProp_Additional_Internal::RecPowerInterNoise***
Pilot: Total received (signal + noise + interference)

- 0 = disabled
- 1 = enabled

RecInterNoise

int ***WinProp_Additional_Internal::RecInterNoise***
Pilot: Interference level (noise + interference)

- 0 = disabled
- 1 = enabled

ReferenceSignals

int ***WinProp_Additional_Internal::ReferenceSignals***
Reference Signals (RSRP, RSSI, RSRQ)

- 0 = disabled

- 1 = enabled

TransTxPower

int ***WinProp_Additional_Internal::TransTxPower***
Minimum required transmit power (transmission mode)

- 0 = disabled
- 1 = enabled

TransRecPower

int ***WinProp_Additional_Internal::TransRecPower***
Maximum achievable received Power (transmission mode)

- 0 = disabled
- 1 = enabled

TransRecProbability

int ***WinProp_Additional_Internal::TransRecProbability***
Reception probability (transmission mode)

- 0 = disabled
- 1 = enabled

TransSNIR

int ***WinProp_Additional_Internal::TransSNIR***
Maximum achievable SNIR (transmission mode)

- 0 = disabled
- 1 = enabled

TransNrParallelStreams

int ***WinProp_Additional_Internal::TransNrParallelStreams***
Maximum number of parallel streams (transmission mode)

- 0 = disabled
- 1 = enabled

TransDataRates

int ***WinProp_Additional_Internal::TransDataRates***
Throughput (transmission mode)

- 0 = disabled
- 1 = enabled

TrafficState

int ***WinProp_Additional_Internal::TrafficState***
Traffic State

- 0 = disabled

- 1 = enabled

TrafficOffered

int ***WinProp_Additional_Internal::TrafficOffered***
Offered Traffic

- 0 = disabled
- 1 = enabled

TrafficServed

int ***WinProp_Additional_Internal::TrafficServed***
Served Traffic

- 0 = disabled
- 1 = enabled

TrafficBlocked

int ***WinProp_Additional_Internal::TrafficBlocked***
Blocked Traffic

- 0 = disabled
- 1 = enabled

NrUsersGenerated

int ***WinProp_Additional_Internal::NrUsersGenerated***
Number of subscribers of application

- 0 = disabled
- 1 = enabled

AirInterfaceFile

WinProp_Filename ***WinProp_Additional_Internal::AirInterfaceFile***
Air interface filename

The documentation was generated from the following file:

- `source/WPAPI2Proj/WPAPI2Proj.h`

2.15.37 WinProp_Antenna

Data Fields

WINPROP_ANTENNA_MODE
Enabled

int
Id

int
SiteId

double
Longitude_X

double
Latitude_Y

double
Height

double
PredictRadius

int
Model

WINPROP_RESULT_TYPE
DataType

int
DataMode

double
Power

WINPROP_POWER_MODE
PowerMode

float
TRXCableLoss

double
Frequency

double
Azimuth

double
Downtilt

WINPROP_ANTENNA_TYPE
Type

double
Gain

WINPROP_ANTENNA_POLARIZATION_MODE
PolarizationMode

float
PolarizationAngle

float
PolarizationXPD

WinProp_Pattern *
Pattern

WinProp_Pattern *
PatternPolHoriz

WinProp_NameString
Name

WINPROP_REPEATER_MODE
RepeaterMode

float
SignalAmpGain

int
NbrCableSegs

COORDPOINT *
CablePoints

double
CableLossPer100m

double
CouplingLoss

double
DistCouplingLoss

double
CouplingGrad

int
RepeaterStatus

int
RepeaterIndex

int
HeightAbsolute

float
RxNoiseFigure

int
GroupIDMounted

const WinProp_Trajectory *
Trajectory

int
NrTimeVariantSets

WinProp_Dynamic_Antenna *
TimeVariantSets

int
SignalMode

float
PowerConstant

WinProp_Filename
PowerResultFile

int
BandType

int
CarrierID

float
LeakageRatio

int
ComponentID

double
RxAdditionalLoss

WinProp_Carrier
Carriers

Detailed Description

Configuration of antenna.

Field Documentation

Enabled

WINPROP_ANTENNA_MODE ***WinProp_Antenna::Enabled***
Use transmitter or not. This parameter is influenced by the optimization module.

- ANTENNA_DISABLED = antenna disabled
- ANTENNA_ENABLED = antenna enabled
- ANTENNA_ENABLED_FIXED = antenna enabled and must not be removed during optimization

Id

int ***WinProp_Antenna::Id***
Identifier of transmitter. The identifier must be greater than zero.

SiteId

int ***WinProp_Antenna::SiteId***
Identifier of site, the transmitter belongs to. The identifier must be greater than zero.

Longitude_X

double ***WinProp_Antenna::Longitude_X***
Longitude or X coordinate of transmitter.

Latitude_Y

double ***WinProp_Antenna::Latitude_Y***
Latitude or Y coordinate of transmitter.

Height

double ***WinProp_Antenna::Height***
Height of transmitter.

PredictRadius

double ***WinProp_Antenna::PredictRadius***
Predict an area around the transmitter.

Model

int ***WinProp_Antenna::Model***
Wave propagation model used:

- WINPROP_MODEL_FREESPACE
- WINPROP_MODEL_COST231
- WINPROP_MODEL_DPM
- WINPROP_MODEL_SRT
- WINPROP_MODEL_IRT (requires a preprocessed vector database [*.idi])

DataType

WINPROP_RESULT_TYPE ***WinProp_Antenna::DataType***
Type of result to be computed:

- PROP_RESULT_POWER
- PROP_RESULT_FIELD
- PROP_RESULT_PATHLOSS

DataMode

int ***WinProp_Antenna::DataMode***
Type of propagation map assigned to this transmitter:

- 0 = prediction result
- 1 = interpolation result

Power

double ***WinProp_Antenna::Power***
Transmit power in dBm.

PowerMode

WINPROP_POWER_MODE ***WinProp_Antenna::PowerMode***
Power mode.

TRXCableLoss

float ***WinProp_Antenna::TRXCableLoss***
Static cable loss (dB) for transmitters not based on components

Frequency

double ***WinProp_Antenna::Frequency***
Frequency in MHz.

Azimuth

double ***WinProp_Antenna::Azimuth***
Azimuth in degrees. North over east orientation.

Downtilt

double ***WinProp_Antenna::Downtilt***
Downtilt in degrees.

Type

WINPROP_ANTENNA_TYPE ***WinProp_Antenna::Type***
Type of antenna.

Gain

double ***WinProp_Antenna::Gain***
Gain of antenna in dBi.

PolarizationMode

WINPROP_ANTENNA_POLARIZATION_MODE ***WinProp_Antenna::PolarizationMode***
Polarization of transmitted signal see ANTENNA_POLARIZATION_MODE_ if non-full polarimetric computation

PolarizationAngle

float ***WinProp_Antenna::PolarizationAngle***
Polarization angle in deg [-90...+90] if PolarizationMode set to ANTENNA_POLARIZATION_MODE_LINEAR_ARBITRARY

- 0 = Vertical
- 90 = Horizontal

PolarizationXPD

float ***WinProp_Antenna::PolarizationXPD***
Cross polarization discrimination in dB

Pattern

WinProp_Pattern * ***WinProp_Antenna::Pattern***
Pattern of antenna total gain in case of standard project or gain for vertical polarization in case of full-polarimetric. If isotropic radiator, this can be NULL.

PatternPolHoriz

WinProp_Pattern * ***WinProp_Antenna::PatternPolHoriz***
Pattern of antenna gain for horizontal polarization in case of full-polarimetric. If isotropic radiator or standard polarimetric analysis, this can be NULL.

Name

WinProp_NameString ***WinProp_Antenna::Name***
Name of antenna. This must be unique.

RepeaterMode

WINPROP_REPEATER_MODE ***WinProp_Antenna::RepeaterMode***
Repeater mode.

SignalAmpGain

float ***WinProp_Antenna::SignalAmpGain***
Signal amplification gain (dB) in case of transparent repeater

NbrCableSegs

int ***WinProp_Antenna::NbrCableSegs***
Number of cable segments (points) in CablePoints, if radiating cable.

CablePoints

COORDPOINT * ***WinProp_Antenna::CablePoints***
Coordinates of points of radiating cable. Needs to be an array of size NbrCableSegs

CableLossPer100m

double ***WinProp_Antenna::CableLossPer100m***
Cable loss for radiating cable per 100 m in dB.

CouplingLoss

double ***WinProp_Antenna::CouplingLoss***
Coupling loss for radiating cable in dB.

DistCouplingLoss

double ***WinProp_Antenna::DistCouplingLoss***
Distance coupling loss (m) for radiating cable.

CouplingGrad

double ***WinProp_Antenna::CouplingGrad***
Gradient Coupling loss for radiating cable per 100 m in dB.

RepeaterStatus

int ***WinProp_Antenna::RepeaterStatus***
Repeater status:

- 0 = Ordinary sector
- 1 = Urban sector which feeds repeaters
- 2 = Repeater in indoor environment
- 3 = outdoor sector (considered as noise only)

RepeaterIndex

int ***WinProp_Antenna::RepeaterIndex***
If repeater (RepeaterStatus is not set to 0), this index contains the unique number of the sector.

HeightAbsolute

int ***WinProp_Antenna::HeightAbsolute***
Height definition of antenna absolute (1) or relative (0).

RxNoiseFigure

float ***WinProp_Antenna::RxNoiseFigure***
Noise figure of receiver [dB]. Only relevant for network planning (see WinProp_Net_Project_Open).

GroupIDMounted

int ***WinProp_Antenna::GroupIDMounted***
In case of a time variant prediction, ID of group where antenna is mounted. The antenna location then is moved with the same vector like the specified group.

Trajectory

const WinProp_Trajectory * ***WinProp_Antenna::Trajectory***
In case of a time variant prediction, the trajectory along which the antenna is moving. The antenna location then is moved with the same vector like the specified trajectory.

NrTimeVariantSets

int ***WinProp_Antenna::NrTimeVariantSets***
In case of a time variant prediction, number of time variant sets for the moving Tx antenna.

TimeVariantSets

WinProp_Dynamic_Antenna * ***WinProp_Antenna::TimeVariantSets***
In case of a time variant prediction, the time variant sets for the moving Tx antenna can be defined including timestep, location, orientation, velocity and moving direction.

SignalMode

int ***WinProp_Antenna::SignalMode***
Signal mode of the external site. Only relevant for external interference sites

- 0 = Constant Signal Power.
- 1 = Signal Power from a Result File.

PowerConstant

float ***WinProp_Antenna::PowerConstant***
Constant interference power level [dBm]. Only relevant for external interference sites with signal mode = 0 (Constant Signal Power).

PowerResultFile

WinProp_Filename ***WinProp_Antenna::PowerResultFile***
File name defining the interference power level. Only relevant for external interference sites with signal mode = 1 (Signal Power from a Result File).

BandType

int ***WinProp_Antenna::BandType***
Type of interference signal. Only relevant for external interference sites.

- 0 = In-band interference (interference due to an external source radiating on a carrier frequency used in the project).
- 1 = Out-of-band interference (interference due to leakage of external signals into frequency bands used in the project).

CarrierID

int ***WinProp_Antenna::CarrierID***
Identifier of the victim carrier. Only relevant for external interference sites with band type = 0 (in-band interference).

LeakageRatio

float ***WinProp_Antenna::LeakageRatio***
Leakage ratio [dB]. Only relevant for external interference sites with band type = 1 (Out-of-band interference).

ComponentID

int ***WinProp_Antenna::ComponentID***
Unique ID of antenna component in case of component based transmitter

RxAdditionalLoss

double ***WinProp_Antenna::RxAdditionalLoss***
Additional loss for UL [dB], required for components

Carriers

WinProp_Carrier ***WinProp_Antenna::Carriers***
Carriers definition, only relevant for network planning

The documentation was generated from the following file:

- `source/Interface/WP_Antenna.h`

2.15.38 WinProp_Antenna_Area

Data Fields

WinProp_Antenna
m_ant

double
m_lowerLeftX

double
m_lowerLeftY

double
m_upperRightX

double
m_upperRightY

Detailed Description

Field Documentation

m_ant

WinProp_Antenna ***WinProp_Antenna_Area::m_ant***

m_lowerLeftX

double ***WinProp_Antenna_Area::m_lowerLeftX***

m_lowerLeftY

double ***WinProp_Antenna_Area::m_lowerLeftY***

m_upperRightX

double ***WinProp_Antenna_Area::m_upperRightX***

m_upperRightY

double ***WinProp_Antenna_Area::m_upperRightY***

WinProp_Antenna_Area::WinProp_Antenna_Area

WinProp_Antenna_Area::WinProp_Antenna_Area

WinProp_Antenna_Area::WinProp_Antenna_Area

WinProp_Antenna_Area::WinProp_Antenna_Area() ***WinProp_Antenna_Area****()*

Description

Parameters

Returns None

WinProp_Antenna_Area::WinProp_Antenna_Area(const ***WinProp_Antenna_Area****(const* WinProp_Antenna *& thisAnt)*

Description

Parameters

const WinProp_Antenna *& thisAnt*

Returns None

WinProp_Antenna_Area::WinProp_Antenna_Area(const ***WinProp_Antenna_Area****(const* WinProp_Antenna *& thisAnt, double llX, double llY, double urX, double urY)*

Description

Parameters

const WinProp_Antenna *& thisAnt*

double llX

double llY

double urX

double urY

Returns None

The documentation was generated from the following file:

- `source/WPAPI2Proj/WPAPI2Proj.h`

2.15.39 WinProp_Antenna_Array

Data Fields

WINPROP_ARRAY_ELEMENT_TYPE
BaseElement

float
Frequency

float
DipoleLength

int
GroundPlane

WinProp_Filename
ExtenalElementFilename

int
NrElementsAxis1

int
NrElementsAxis2

float
ElementSpacingAxis1

float
ElementSpacingAxis2

float
PhaseShiftAxis1

float
PhaseShiftAxis2

WINPROP_ARRAY_AMPLITUE_TAPER
AmplitudeTaper

float
DolphChebyshevSSL

WINPROP_ARRAY_PLANE
AntennaArrayPlane

WinProp_Filename
OutputFilename

Detailed Description

This structure is used to set the parameters for generating a beam pattern using linear/planar phased array antenna when calling WinProp_GenerateAntennaArrayPattern or for network planning beam steering.

Field Documentation

BaseElement

WINPROP_ARRAY_ELEMENT_TYPE ***WinProp_Antenna_Array::BaseElement***
Type of the base element antenna:

- WINPROP_ARRAY_ELEMENT_TYPE::ARRAY_ELEMENT_ISOTROPIC = Isotropic antenna element
- WINPROP_ARRAY_ELEMENT_TYPE::ARRAY_ELEMENT_DIPOLE = Dipole antenna element
- WINPROP_ARRAY_ELEMENT_TYPE::ARRAY_ELEMENT_EXTERNAL = External antenna element

Frequency

float ***WinProp_Antenna_Array::Frequency***
Frequency in MHz of the base element

DipoleLength

float ***WinProp_Antenna_Array::DipoleLength***
Length of the dipole antenna in wavelengths. Only relevant for BaseElement = WINPROP_ARRAY_ELEMENT_TYPE::ARRAY_ELEMENT_DIPOLE.

GroundPlane

int ***WinProp_Antenna_Array::GroundPlane***
Consider a perfectly conducting ground plane

- 0 = No
- 1 = yes

ExtenalElementFilename

WinProp_Filename ***WinProp_Antenna_Array::ExtenalElementFilename***
Filename of the external antenna element. Only relevant for BaseElement = WINPROP_ARRAY_ELEMENT_TYPE::ARRAY_ELEMENT_EXTERNAL.

NrElementsAxis1

int ***WinProp_Antenna_Array::NrElementsAxis1***
Number of the base elements along the first axis.

NrElementsAxis2

int ***WinProp_Antenna_Array::NrElementsAxis2***
Number of the base elements along the second axis.

ElementSpacingAxis1

float ***WinProp_Antenna_Array::ElementSpacingAxis1***
Spacing offset along the first axis in wavelengths.

ElementSpacingAxis2

float ***WinProp_Antenna_Array::ElementSpacingAxis2***
Spacing offset along the second axis in wavelengths.

PhaseShiftAxis1

float ***WinProp_Antenna_Array::PhaseShiftAxis1***
Phase shift between elements along the first axis in degrees.

PhaseShiftAxis2

float ***WinProp_Antenna_Array::PhaseShiftAxis2***
Phase shift between elements along the second axis in degrees.

AmplitudeTaper

WINPROP_ARRAY_AMPLITUE_TAPER ***WinProp_Antenna_Array::AmplitudeTaper***
Type of the amplitude taper:

- WINPROP_ARRAY_AMPLITUE_TAPER::ARRAY_AMPLITUE_TAPER_UNIFORM = Uniform amplitude
- WINPROP_ARRAY_AMPLITUE_TAPER::ARRAY_AMPLITUE_TAPER_TRAINGLE = Triangular amplitude taper
- WINPROP_ARRAY_AMPLITUE_TAPER::ARRAY_AMPLITUE_TAPER_COSINE = Cosine amplitude taper
- WINPROP_ARRAY_AMPLITUE_TAPER::ARRAY_AMPLITUE_TAPER_DOLPH_CHEBYSHEV = Dolph-Chebyshev amplitude taper

DolphChebyshevSSL

float ***WinProp_Antenna_Array::DolphChebyshevSSL***
Required sidelobe suppression level (SSL) in dB for the Dolph-Chebyshev amplitude taper. Needs to be greater than zero. Only relevant for AmplitudeTaper = WINPROP_ARRAY_AMPLITUE_TAPER::ARRAY_AMPLITUE_TAPER_DOLPH_CHEBYSHEV.

AntennaArrayPlane

WINPROP_ARRAY_PLANE ***WinProp_Antenna_Array::AntennaArrayPlane***
The plane on which the antenna array lies:

- WINPROP_ARRAY_PLANE::ARRAY_PLANE_XY = XY-Plane
- WINPROP_ARRAY_PLANE::ARRAY_PLANE_XZ = XZ-Plane

OutputFilename

WinProp_Filename ***WinProp_Antenna_Array::OutputFilename***
Filename of output file without extension.

The documentation was generated from the following file:

- `source/Beams/winprop_beams_types.h`

2.15.40 WinProp_Area

Data Fields

double
LowerLeftX

double
LowerLeftY

double
UpperRightX

double
UpperRightY

double
Resolution

int
NrHeights

*double **
Heights

*void **
CoordSystem

int
HeightAbsolute

Detailed Description

This structure is used for the definition of the prediction area. It is passed to the function WinProp_Predict.

Field Documentation

LowerLeftX

double ***WinProp_Area::LowerLeftX***
Lower left corner (X coordinate) of prediction area.

LowerLeftY

double ***WinProp_Area::LowerLeftY***
Lower left corner (Y coordinate) of prediction area.

UpperRightX

double ***WinProp_Area::UpperRightX***
Upper right corner (X coordinate) of prediction area.

UpperRightY

double ***WinProp_Area::UpperRightY***
Upper right corner (Y coordinate) of prediction area.

Resolution

double ***WinProp_Area::Resolution***
Resolution (meter).

NrHeights

int ***WinProp_Area::NrHeights***
Number of prediction heights. Heights needs to be an array of this size.

Heights

double * ***WinProp_Area::Heights***
Array with prediction heights (meter).See also NrHeights

CoordSystem

void * ***WinProp_Area::CoordSystem***
Information about coordinate system. Set to NULL.

HeightAbsolute

int ***WinProp_Area::HeightAbsolute***
Height definition of receiving antenna.

- 0 = Height relative to ground
- 1 = Height absolute to sea level
- 2 = Height relative to floors

The documentation was generated from the following file:

- `source/Interface/WP_Area.h`

2.15.41 WinProp_AreaPlane

Data Fields

int
NrCorners

COORDPOINT *
Corners

double
Resolution

void *
CoordSystem

Detailed Description

This structure is used for the definition of a prediction area on a polygon. It is passed to the function WinProp_Predict_Planes.

Field Documentation

NrCorners

int ***WinProp_AreaPlane::NrCorners***
Number of corners (at least 3)

Corners

COORDPOINT * ***WinProp_AreaPlane::Corners***
List with corners of polygon (X, Y, Z in meter)

Resolution

double ***WinProp_AreaPlane::Resolution***
Resolution on plane (meter)

CoordSystem

void * ***WinProp_AreaPlane::CoordSystem***
Information about coordinate system. Set to NULL.

The documentation was generated from the following file:

- `source/Interface/WP_Area.h`

2.15.42 WinProp_Calib_DPM

Data Fields

WINPROP_SCENARIO
Scenario

int
NbrTuningFiles

WinProp_Filename *
TuningFiles

double
MinDistance

double
MaxDistance

double
MinPathLoss

double
MaxPathLoss

double
MinPower

double
MaxPower

double
ExpLOS

double
ExpOLOS

double
ExpNLOS

double
ExpLOSafterBP

double
ExpOLOSafterBP

int
NbrLOS

int
NbrOLOS

int
NbrNLOS

int
NbrLOSafterBP

int
NbrOLOSafterBP

double
DiffrLoss

double
Offset

double
Offset_LOS

double
Offset_NonLOS

Detailed Description

This structure is used to calibrate the parameters of the DPM (Dominant Path Model). The structure can be passed to the function WinProp_Calibrate_DPM, but this is optional.

Field Documentation

Scenario

WINPROP_SCENARIO ***WinProp_Calib_DPM::Scenario***
Type of scenario (urban, indoor, rural)

NbrTuningFiles

int ***WinProp_Calib_DPM::NbrTuningFiles***
Number of calibration files. Calibration files are generated during a prediction with the DPM (see WinProp_Predict)

TuningFiles

WinProp_Filename * ***WinProp_Calib_DPM::TuningFiles***
Array with file names of calibration files

MinDistance

double ***WinProp_Calib_DPM::MinDistance***
Minimum required distance from transmitter to measurement point

MaxDistance

double ***WinProp_Calib_DPM::MaxDistance***
Maximum required distance from transmitter to measurement point

MinPathLoss

double ***WinProp_Calib_DPM::MinPathLoss***
Minimum required path loss of a measurement point

MaxPathLoss

double ***WinProp_Calib_DPM::MaxPathLoss***
Maximum required path loss of a measurement point

MinPower

double ***WinProp_Calib_DPM::MinPower***
Minimum required power (dBm) of a measurement point

MaxPower

double ***WinProp_Calib_DPM::MaxPower***
Maximum required power (dBm) of a measurement point

ExpLOS

double ***WinProp_Calib_DPM::ExpLOS***
Calibrated LOS exponent set by WinProp_Calibrate_DPM

ExpOLOS

double ***WinProp_Calib_DPM::ExpOLOS***
Calibrated OLOS exponent set by WinProp_Calibrate_DPM

ExpNLOS

double ***WinProp_Calib_DPM::ExpNLOS***
Calibrated NLOS exponent set by WinProp_Calibrate_DPM

ExpLOSafterBP

double ***WinProp_Calib_DPM::ExpLOSafterBP***
Calibrated LOS2 exponent set by WinProp_Calibrate_DPM for urban scenarios onlySee also Scenario

ExpOLOSafterBP

double ***WinProp_Calib_DPM::ExpOLOSafterBP***
Calibrated OLOS2 exponent set by WinProp_Calibrate_DPM for urban scenarios onlySee also Scenario

NbrLOS

int ***WinProp_Calib_DPM::NbrLOS***
Calibration result: Number of LOS pixels considered during the calibration (set by WinProp_Calibrate_DPM)

NbrOLOS

int ***WinProp_Calib_DPM::NbrOLOS***
Calibration result: Number of OLOS pixels considered during the calibration (set by WinProp_Calibrate_DPM)

NbrNLOS

int ***WinProp_Calib_DPM::NbrNLOS***
Calibration result: Number of NLOS pixels considered during the calibration (set by WinProp_Calibrate_DPM)

NbrLOSafterBP

int ***WinProp_Calib_DPM::NbrLOSafterBP***
Calibration result: Number of LOS pixels after breakpoint considered during the calibration. Set by WinProp_Calibrate_DPM for urban scenarios only.

NbrOLOSafterBP

int ***WinProp_Calib_DPM::NbrOLOSafterBP***
Calibration result: Number of OLOS pixels after breakpoint considered during the calibration. Set WinProp_Calibrate_DPM for urban scenarios only

DiffrLoss

double ***WinProp_Calib_DPM::DiffrLoss***
Calibrated diffraction loss set by WinProp_Calibrate_DPM

Offset

double ***WinProp_Calib_DPM::Offset***
Calibrated offset for all pixels. Set by WinProp_Calibrate_DPM

Offset_LOS

double ***WinProp_Calib_DPM::Offset_LOS***
Offset for LOS pixels set by WinProp_Calibrate_DPM

Offset_NonLOS

double ***WinProp_Calib_DPM::Offset_NonLOS***
Offset for NLOS/OLOS pixels set by WinProp_Calibrate_DPM

The documentation was generated from the following file:

- `source/Interface/WP_Measurements.h`

2.15.43 WinProp_Callback

Data Fields

CALLBACK_PERCENTAGE
: Percentage

CALLBACK_MESSAGEINFO
: Message

CALLBACK_ERROR
: Error

CALLBACK_TERMINATE
: TermComp

unsigned int
: Synchronize

Detailed Description

This structure is used to set the callback functions when calling WinProp_Open.

Field Documentation

Percentage

CALLBACK_PERCENTAGE ***WinProp_Callback::Percentage***
: Function pointer to function called for progress bar update.

Message

CALLBACK_MESSAGEINFO ***WinProp_Callback::Message***
: Function pointer to function called for message output.

Error

CALLBACK_ERROR ***WinProp_Callback::Error***
: Function pointer to function called for error output.

TermComp

CALLBACK_TERMINATE ***WinProp_Callback::TermComp***
: Function pointer to function called for termination request.

Synchronize

unsigned int ***WinProp_Callback::Synchronize***
: Unless set to zero, multiple threads will synchronize when calling the callbacks.

The documentation was generated from the following file:

- `source/Interface/WP_Callback.h`

2.15.44 WinProp_Carrier

Data Fields

int
CarrierID

int
SystemID

int
MimoID

float
CellLoad

float
NoiseRiseUL

float
HandoverOffsetdB

float
HandoverTTTms

float
PowerBackoffPilotDL

int
Numerology

int
SymbolsPerSlot

int
NrAntennaBeams

double
BeamGainCtrlServer

double
BeamGainCtrlInterferer

double
BeamGainDataServer

double
BeamGainDataInterferer

int
TDD_Slots_DL

int
TDD_Slots_UL

int
TDD_Slots_Flex

Detailed Description

Configuration of Carriers.

Field Documentation

CarrierID

int ***WinProp_Carrier::CarrierID***
Index of carrier (if network planning API is used afterwards).

SystemID

int ***WinProp_Carrier::SystemID***
Index of system to which the antenna belongs. If SystemID == -1, the antenna does not belong a system.

- Same carrier, different SystemID => Interference
- Same carrier, same SystemID => No interference
- Different carriers (independent of SystemID) => No interference

MimoID

int ***WinProp_Carrier::MimoID***
Index of MIMO stream assigned to this antenna. If MimoID == -1, it is a regular antenna. Antennas of MIMO systems must be in the same system, i.e. SystemID must be set to the same value for all antennas belonging to the MIMO system. The first antenna in a MIMO system has MimoID == 1, the second antenna has MimoID == 2 and so on.

CellLoad

float ***WinProp_Carrier::CellLoad***
Load of this cell, only relevant for network planning (see WinProp_Net_Project_Open). Set to -1 to use the default load. Needs to be in the interval [0.0, 1.0].

NoiseRiseUL

float ***WinProp_Carrier::NoiseRiseUL***
The noise rise [dB] in uplink for static analysis, only relevant for network planning (see WinProp_Net_Project_Open). Set to -1 to use the default value.

HandoverOffsetdB

float ***WinProp_Carrier::HandoverOffsetdB***
The handover offset [dB], only relevant for network planning (see WinProp_Net_Project_Open). Set to -1 to use the default value.

HandoverTTTms

float ***WinProp_Carrier::HandoverTTTms***
The handover time to trigger (TTT) in [ms], only relevant for network planning (see WinProp_Net_Project_Open). Set to -1 to use the default value.

PowerBackoffPilotDL

float ***WinProp_Carrier::PowerBackoffPilotDL***
The power backoff for pilot [dB], relative to max. power. Only relevant for network planning (see WinProp_Net_Project_Open). Set to -1 to use the default value.

Numerology

int ***WinProp_Carrier::Numerology***
The numerology. Only relevant for 5G network planning.

SymbolsPerSlot

int ***WinProp_Carrier::SymbolsPerSlot***
Number of symbols per slot. Only relevant for 5G network planning.

NrAntennaBeams

int ***WinProp_Carrier::NrAntennaBeams***
Number of antenna beams. Only relevant for 5G network planning.

BeamGainCtrlServer

double ***WinProp_Carrier::BeamGainCtrlServer***
Beamforming Gain for control serving carrier. Only relevant for 5G network planning.

BeamGainCtrlInterferer

double ***WinProp_Carrier::BeamGainCtrlInterferer***
Beamforming Gain for control interfering carrier. Only relevant for 5G network planning.

BeamGainDataServer

double ***WinProp_Carrier::BeamGainDataServer***
Beamforming Gain for data serving carrier. Only relevant for 5G network planning.

BeamGainDataInterferer

double ***WinProp_Carrier::BeamGainDataInterferer***
Beamforming Gain for data interfering carrier. Only relevant for 5G network planning.

TDD_Slots_DL

int ***WinProp_Carrier::TDD_Slots_DL***
Number of downlink symbols in a slot. Only relevant for TDD 5G network planning.

TDD_Slots_UL

int ***WinProp_Carrier::TDD_Slots_UL***
Number of uplink symbols in a slot. Only relevant for TDD 5G network planning.

TDD_Slots_Flex

int ***WinProp_Carrier::TDD_Slots_Flex***
Number of flexible symbols in a slot. Only relevant for TDD 5G network planning.

The documentation was generated from the following file:

- `source/Interface/WP_Carrier.h`

2.15.45 WinProp_Convert_Beams

Data Fields

WinProp_Filename
OutputFilename

int
GenerateEnvelopeBeams

int
GenerateEnvelope

int
NrBeamsCtrl

int
NrBeamsData

WinProp_Filename *
FilenamesCtrl

WinProp_Filename *
FilenamesData

int
LimitDynamicRange

float
DynamicRange

float
Frequency

float
PolAngle

float
XPD

Detailed Description

This structure is used to set the parameters for converting/combining a set of individual beams (available in a set of individual antenna-pattern files) into envelope and individual beam patterns when calling WinProp_ConvertBeams.

Field Documentation

OutputFilename

WinProp_Filename ***WinProp_Convert_Beams::OutputFilename***

Filename of output files without extension. For the file of the envelope and beam patterns, "_BeamPatterns" is appended to the filename. For the file of the envelope patterns, "_EnvelopePattern" is appended to the filename.

GenerateEnvelopeBeams

int ***WinProp_Convert_Beams::GenerateEnvelopeBeams***

Generate envelope patterns and a set of individual beams:

- 0 = No
- 1 = yes

GenerateEnvelope

int ***WinProp_Convert_Beams::GenerateEnvelope***

Generate only envelope in a separate file:

- 0 = No
- 1 = yes

NrBeamsCtrl

int ***WinProp_Convert_Beams::NrBeamsCtrl***

Number of control beams to convert.

NrBeamsData

int ***WinProp_Convert_Beams::NrBeamsData***

Number of data beams to convert.

FilenamesCtrl

WinProp_Filename * ***WinProp_Convert_Beams::FilenamesCtrl***

Array of filenames of the control beams to convert. Needs to be an array of size NrBeamsCtrl .

FilenamesData

WinProp_Filename * ***WinProp_Convert_Beams::FilenamesData***

Array of filenames of the data beams to convert. Needs to be an array of size NrBeamsData .

LimitDynamicRange

int ***WinProp_Convert_Beams::LimitDynamicRange***
Limit dynamic range of the antenna pattern:

- 0 = No
- 1 = yes

DynamicRange

float ***WinProp_Convert_Beams::DynamicRange***
Dynamic range of the antenna pattern in dB. Only relevant if LimitDynamicRange is enabled.

Frequency

float ***WinProp_Convert_Beams::Frequency***
Frequency in MHz. Relevant only when converting non *.ffe files.

PolAngle

float ***WinProp_Convert_Beams::PolAngle***
Linear Polarization angle in degrees. Relevant only when converting non *.ffe files.

XPD

float ***WinProp_Convert_Beams::XPD***
Cross-polarization discrimination in dB. Relevant only when converting non *.ffe files.

The documentation was generated from the following file:

- `source/Beams/winprop_beams_types.h`

2.15.46 WinProp_Converter

Data Fields

float
PixelResolution

int
UndefinedPixelMode

float
UndefinedPixelValue

float
UndefinedPixelWindowSize

float
BuildingLevelHeightOSM

int
NrBuildingLevelsOSM

float
RoofLevelHeightOSM

int
BuildingHeightMode

float
BuildingHeightDefault

int
ConverterID

*const char **
databaseNameSource

*const char **
databaseNameDest

*const char **
measurementUnit

*char ***
ignoreList

int
nbrElements

double
scalingFactor

int
InCoordGeoUTM

int
OutCoordGeoUTM

int
Ellipsoid

int
UTMAuto

int
UTMZoneNr

int
UTMNorth

int
ExtendedMessageOutput

int
SaveAsciiFormat

int
ConversionMethod

float
DiscretizationStep

Detailed Description

General Data for calling converter modules in the API, specifies the conversion method and the in- and output files. The ConverterID selects the conversion method and specifies the input format. It can be one of the following:

Topography

- 1: ASCII Line Format
- 2: WinProp Format (*.tdb)
- 3: WinProp Format Index File (*.tdi)
- 4: ASCII Grid Format (*.asc)
- 5: Binary Grid Format (*.bin)
- 6: Nokia NPS/X (index data file)
- 7: Nokia NPS/X (single data file)
- 8: MSI Planet / Siemens Tornado (index data file)
- 9: MSI Planet / Siemens Tornado (single data file)
- 11: Aircom ASSET / NSN NetAct
- 12: Agilent Wizard
- 13: Planetary Data System v3/v4
- 14: USGS DEM
- 15: USGS BIL
- 16: HGT (3 ARC)
- 17: HGT (1 ARC)
- 18: Digital Terrain Elevation Data
- 19: GeoTIFF

Morpho / Clutter

- 101: ASCII Line Format
- 102: WinProp Format
- 104: ASCII Grid Format (*.asc)
- 105: Binary Grid Format (*.bin)
- 106: Nokia NPS/X (index data file)
- 107: Nokia NPS/X (single data file)
- 108: MSI Planet / Siemens Tornado (index data file)
- 109: MSI Planet / Siemens Tornado (single data file)
- 111: Aircom ASSET / NSN NetAct
- 112: Agilent Wizard
- 113: GeoTIFF

Urban

- 201: WinProp Urban Buildings ASCII (*.oda)
- 202: Vector Data ASCII Format (*.vda)
- 203: Geography Markup Language (*.gml)
- 204: MapInfo File (*mif, *.tab)
- 205: Arcview Shapefile (*.shp)
- 206: AutoCAD (*.dwg, *.dxf)
- 207: VRML Format
- 208: Aircom Asset / NSN NetAct (single file)
- 209: Aircom Asset / NSN NetAct (index file)
- 210: Agilent Wizard Building Data
- 211: MSI Planet Building Data (single file)
- 212: Gen Building Data Format (*.gen)
- 213: Building Database Format (D2 FUN)
- 214: Pegasos Building Data Format (*.geb)
- 215: MCS File Format
- 216: Open Street Map (*.osm)
- 217: Urban Building Binary (*.odb) -> Indoor Building Binary (*.idb)
- 218: CityGML (*.gml, *.citygml) [Only available for Windows version]

Indoor

- 301: WinProp Indoor Building Data ASCII (*.ida)
- 302: AutoCAD File Format (*.dwg, *.dxf)
- 303: Stereolithography File Format (*.stl) (ASCII/Binary)
- 304: Stereolithography File Format (*.stl) (ASCII)
- 305: Stereolithography File Format (*.stl) (Binary)
- 306: Nastran File Format (*.nas)
- 307: Wavefront File Format (*.obj)
- 308: Facet File Format (*.fac)
- 309: MCS File Format
- 310: Building Information Modeling File Format (*.ifc)
- 311: Geography Markup Language (*.gml)
- 312: Autodesk 3ds Max File Format (*.3ds)
- 313: Autodesk Filmbox File Format (*.fbx)
- 314: Blender File Format (*.blend)
- 315: GL Transmission Format (*.gltf, *.glb)
- 316: COLLADA (*.dae)
- 317: Indoor Building Binary (*.idb) -> Urban Building Binary (*.odb)
- 318: CityGML (*.gml, *.citygml) [Only available for Windows version]
- 319: OpenDRIVE (*.xodr)

Field Documentation

PixelResolution

float ***WinProp_Converter::PixelResolution***
Resolution for pixels (for ASCII Line format topo/clutter only):

- UTM: default 10m
- Geodetic: default 0.001Â°

UndefinedPixelMode

int ***WinProp_Converter::UndefinedPixelMode***
How to handle undefined pixels (for topo/clutter only):

- 0: Remain undefined
- 1: Set to constant value
- 2: Interpolate with window

UndefinedPixelValue

float ***WinProp_Converter::UndefinedPixelValue***
Value for undefined pixels (only if UndefinedPixelMode set to 1)

UndefinedPixelWindowSize

float ***WinProp_Converter::UndefinedPixelWindowSize***
Window size [m] for interpolation of undefined pixels (only if UndefinedPixelMode set to 2)

BuildingLevelHeightOSM

float ***WinProp_Converter::BuildingLevelHeightOSM***
Default height of a building level (floor) in [m] (only relevant for urban Open Street Map conversions when an imported building without height information)

NrBuildingLevelsOSM

int ***WinProp_Converter::NrBuildingLevelsOSM***
Default number of building levels (floors) (only relevant for urban Open Street Map conversions when an imported building without height information)

RoofLevelHeightOSM

float ***WinProp_Converter::RoofLevelHeightOSM***
Default height of a roof level in [m] (only relevant for urban Open Street Map conversions when an imported building without height information)

BuildingHeightMode

int ***WinProp_Converter::BuildingHeightMode***
Height mode (only relevant for conversion of some 2.5D data formats), depending on this setting, the height of the buildings will be ...

- 0: the value the property, set within CALLBACK_HEIGHT_LAYER_SELECTION, has for a building.

- 1: the z-value of the building polygons if available, otherwise the value in BuildingHeightDefault
- 2: the value set in BuildingHeightDefault

BuildingHeightDefault

float ***WinProp_Converter::BuildingHeightDefault***
The building default height for cases described for BuildingHeightMode

ConverterID

int ***WinProp_Converter::ConverterID***
ID of converter module (see description above)

databaseNameSource

*const char ** ***WinProp_Converter::databaseNameSource***
Name of original database.

databaseNameDest

*const char ** ***WinProp_Converter::databaseNameDest***
Name of converted database without extension.

measurementUnit

*const char ** ***WinProp_Converter::measurementUnit***
Unit of database (indoor only)

ignoreList

*char *** ***WinProp_Converter::ignoreList***
List with names of ignored AutoCAD layers (indoor only)

nbrElements

int ***WinProp_Converter::nbrElements***
Number of elements in the list of ignored AutoCAD layers

scalingFactor

double ***WinProp_Converter::scalingFactor***
Scaling factor for database

- Indoor: applied to all coordinates, set to negative value for automatic scaling when converting DXF/DWG files
- Urban: applied to building heights
- Rural: applied to topo heights

InCoordGeoUTM

int ***WinProp_Converter::InCoordGeoUTM***
Input map format:

- 0: Geodetic

- 1: UTM

OutCoordGeoUTM

int ***WinProp_Converter::OutCoordGeoUTM***
Converted map format (for topo/clutter only):

- 0: Geodetic
- 1: UTM

Ellipsoid

int ***WinProp_Converter::Ellipsoid***
Input map: Number of Ellipsoid (see the Ellipsoid coordinate defines.)

UTMAuto

int ***WinProp_Converter::UTMAuto***
Autodetection of UTM zone

- 0: manual definition
- 1: auto detection

UTMZoneNr

int ***WinProp_Converter::UTMZoneNr***
UTM zone number (only relevant for manual definition)

UTMNorth

int ***WinProp_Converter::UTMNorth***
UTM zone hemisphere (only relevant for manual definition)

- 0: South
- 1: North

ExtendedMessageOutput

int ***WinProp_Converter::ExtendedMessageOutput***
Extended message output:

- 0 = print less messages
- 1 = print messages

SaveAsciiFormat

int ***WinProp_Converter::SaveAsciiFormat***
Save output database in Binary or ASCII format:

- 0 = Binary format
- 1 = ASCII format

ConversionMethod

int ***WinProp_Converter::ConversionMethod***
Conversion method for roads (only relevant for indoor OpenDRIVE conversions):

- 0 = Auto: converts planar roads with the outline method, roads with elevation profiles with the mesh method
- 1 = Mesh: creates meshed triangular objects for the road lanes
- 2 = Outline: creates walls objects from the contour of the road lanes

DiscretizationStep

float ***WinProp_Converter::DiscretizationStep***
Minimum discretization length in [m]. Larger values will be used where appropriate (only relevant for indoor OpenDRIVE conversions)

The documentation was generated from the following file:

- `source/Interface/EngineConvert.h`

2.15.47 WinProp_Coverage_Para

Data Fields

int
UseFrequency

double
Frequency

int
NeglectIndoorPixels

int
IsUrbanPrediction

Detailed Description

This structure is used to configure the algorithm of WinProp_Coverage_Analysis.

Field Documentation

UseFrequency

int ***WinProp_Coverage_Para::UseFrequency***
Consider a frequency dependency.

Frequency

double ***WinProp_Coverage_Para::Frequency***
Frequency (MHz) if "UseFrequency" is set.

NeglectIndoorPixels

int ***WinProp_Coverage_Para::NeglectIndoorPixels***
Decision parameter to neglect indoor pixels

- 0 = Ignore indoor pixels

- 1 = Consider indoor pixels

IsUrbanPrediction

int ***WinProp_Coverage_Para::IsUrbanPrediction***
Decision parameter to neglect urban pixels. i.e., whether or not they lie inside the measurement polygon

- 0 = Ignore urban pixels
- 1 = Consider urban pixels

The documentation was generated from the following file:

- `source/Interface/WP_Result.h`

2.15.48 WinProp_Dynamic_Antenna

Data Fields

COORDPOINT
Location

COORDPOINT
MovingDir

float
Azimuth

float
Downtilt

float
Roll

float
TimeStep

float
Velocity

Detailed Description

Configuration of dynamic antenna.

Field Documentation

Location

COORDPOINT ***WinProp_Dynamic_Antenna::Location***
Location of Tx antenna at defined timestep

MovingDir

COORDPOINT ***WinProp_Dynamic_Antenna::MovingDir***
Moving direction of Tx antenna at defined timestep

Azimuth

float ***WinProp_Dynamic_Antenna::Azimuth***
Azimuth in degrees at defined timestep. North over east orientation.

Downtilt

float ***WinProp_Dynamic_Antenna::Downtilt***
Downtilt in degrees at defined timestep.

Roll

float ***WinProp_Dynamic_Antenna::Roll***
Roll-angle in degrees at defined timestep.

TimeStep

float ***WinProp_Dynamic_Antenna::TimeStep***
Timestep for which these Tx properties are valid.

Velocity

float ***WinProp_Dynamic_Antenna::Velocity***
Moving direction of Tx antenna at defined timestep.

The documentation was generated from the following file:

- `source/Interface/WP_Antenna.h`

2.15.49 WinProp_Dynamic_Sets

Data Fields

int
NrSets

WinProp_Dynamic_Antenna *
Sets

Detailed Description

This structure is used to define the list of dynamic sets for a given moving group

Field Documentation

NrSets

int ***WinProp_Dynamic_Sets::NrSets***
Number of dynamic sets

Sets

WinProp_Dynamic_Antenna * ***WinProp_Dynamic_Sets::Sets***
Dynamic sets parameters

The documentation was generated from the following file:

- `source/Interface/WP_Propagation.h`

2.15.50 WinProp_FMCW_Para

Data Fields

double
Tc

unsigned int
NrChirps

double
B

FreqBins
NrBins

double
Vmax

double
Vres

double
D

double
Dres

int
NrRx

double
RxSpacing

int
NrTx

double
TxSpacing

Windowing
    WindFunc

bool
    GenerateRangeDoppler

PlotType
    RangeDopplerPlotType

bool
    GenerateRangeAngle

PlotType
    RangeAnglePlotType

bool
    GenerteIQ

WinProp_Filename
    OutputPathIQ

ParametersMode
    Mode

double
    Freq

ElementsConfig
    RayElmsConfig

bool
    Coherent

Detailed Description

This structure contains the FMCW Radar parameters. It is used to pass FMCW radar settings to WinProp_FMCW_Radar::compute.

Field Documentation

Tc

double ***WinProp_FMCW_Para::Tc***
    FMCW chirp period in s

NrChirps

unsigned int ***WinProp_FMCW_Para::NrChirps***
    Number of chirps.

B

double ***WinProp_FMCW_Para::B***
    Sweep bandwidth B in Hz

NrBins

FreqBins ***WinProp_FMCW_Para::NrBins***
Number of frequency bins (range domain) see FreqBins

Vmax

double ***WinProp_FMCW_Para::Vmax***
Maximum velocity in m/s

Vres

double ***WinProp_FMCW_Para::Vres***
Velocity resolution in m/s

D

double ***WinProp_FMCW_Para::D***
Range of interest in m

Dres

double ***WinProp_FMCW_Para::Dres***
Range resolution in m

NrRx

int ***WinProp_FMCW_Para::NrRx***
Number receiving array elements (for angle estimation)

RxSpacing

double ***WinProp_FMCW_Para::RxSpacing***
Spacing between receiving elements in m (for angle estimation)

NrTx

int ***WinProp_FMCW_Para::NrTx***
Number transmitting array elements (for angle estimation)

TxSpacing

double ***WinProp_FMCW_Para::TxSpacing***
Spacing between transmitting elements in m (for angle estimation)

WindFunc

Windowing ***WinProp_FMCW_Para::WindFunc***
Windowing Function see Windowing

GenerateRangeDoppler

bool ***WinProp_FMCW_Para::GenerateRangeDoppler***
True to generate Range-Doppler plot

RangeDopplerPlotType

PlotType ***WinProp_FMCW_Para::RangeDopplerPlotType***
Range-Doppler output plot type

GenerateRangeAngle

bool ***WinProp_FMCW_Para::GenerateRangeAngle***
True to generate Range-Angle plot

RangeAnglePlotType

PlotType ***WinProp_FMCW_Para::RangeAnglePlotType***
Range-Angle output plot type

GenerteIQ

bool ***WinProp_FMCW_Para::GenerteIQ***
True to generate IQ signal (at the ADC output)

OutputPathIQ

WinProp_Filename ***WinProp_FMCW_Para::OutputPathIQ***
Filepath of the IQ ASCII file. Relevant if GenerteIQ is enabled

Mode

ParametersMode ***WinProp_FMCW_Para::Mode***
Parameter mode see ParametersMode

Freq

double ***WinProp_FMCW_Para::Freq***
FMCW radar frequency in Hz

RayElmsConfig

ElementsConfig ***WinProp_FMCW_Para::RayElmsConfig***
The configuration of the ray elements provided when calling WinProp_FMCW_Radar::compute. see ElementsConfig

Coherent

bool ***WinProp_FMCW_Para::Coherent***
True for coherent superposition of rays, false for incoherent superposition of rays (power sum)

WinProp_FMCW_Para::WinProp_FMCW_Para
constructor

WinProp_FMCW_Para::~WinProp_FMCW_Para
Destructor

WinProp_FMCW_Para::WinProp_FMCW_Para
Copy constructor

WinProp_FMCW_Para::WinProp_FMCW_Para() ***WinProp_FMCW_Para****()*

Description

constructor

Parameters

Returns None

WinProp_FMCW_Para::~WinProp_FMCW_Para() ***~WinProp_FMCW_Para****()*

Description

Destructor

Parameters

Returns None

WinProp_FMCW_Para::WinProp_FMCW_Para(const ***WinProp_FMCW_Para****(const* WinProp_FMCW_Para *& other)*

Description

Copy constructor

Parameters

const WinProp_FMCW_Para *& other*

Returns None

Enums

unsigned char ParametersMode

- DESIGN_PARAMETERS
- SET_PARAMETERS An enum constant representing the design parameters option

unsigned char FreqBins

- FFT_8
- FFT_16
- FFT_32
- FFT_64
- FFT_128
- FFT_256
- FFT_512
- FFT_1024
- FFT_2048
- FFT_COUNT

unsigned char Windowing

- NO_WINDOWING
- HANNING An enum constant representing the no windowing option

- HAMMING An enum constant representing the hanning window option
- BLACKMAN An enum constant representing the hamming window option
- BARTLETT An enum constant representing the blackman window option

unsigned char ElementsConfig

- MULTIPLE_ELEMENTS
- MULTIPLE_TIME_STEPS An enum constant representing the multiple receiving or transmitting elements option

unsigned char PlotType

- HEAT_MAP
- CFAR_DETECTION An enum constant representing the FFT heatmap plot type option
- MUSIC_SPECTRUM An enum constant representing the CFAR detection of FFT heatmap plot option
- MUSIC_CFAR An enum constant representing the MUSIC spectrum plot type option (relevant for Range-Angle plots)

The documentation was generated from the following file:

- `source/FMCW/winprop_fmcw_types.h`

2.15.51 WinProp_Generate_Beams

Data Fields

int
NbrAzimuthBeamsCtrl

int
NbrElevationBeamsCtrl

float
AzimuthRangeCtrl

float
ElevationRangeCtrl

float *
MaxGainCtrl

float *
HorBeamwidthCtrl

float *
VerBeamwidthCtrl

int
NbrAzimuthBeamsData

int
NbrElevationBeamsData

float
AzimuthRangeData

float
ElevationRangeData

float *
MaxGainData

float *
HorBeamwidthData

float *
VerBeamwidthData

float
Frequency

float
PolAngle

float
XPD

WinProp_Filename
OutputFilename

int
GenerateEnvelopeBeams

int
GenerateEnvelope

Detailed Description

This structure is used to set the parameters for generating envelope and individual beam patterns when calling WinProp_GenerateBeams.

Field Documentation

NbrAzimuthBeamsCtrl

int ***WinProp_Generate_Beams::NbrAzimuthBeamsCtrl***
Number control beams in the azimuth domain.

NbrElevationBeamsCtrl

int ***WinProp_Generate_Beams::NbrElevationBeamsCtrl***
Number control beams in the elevation domain.

AzimuthRangeCtrl

float ***WinProp_Generate_Beams::AzimuthRangeCtrl***
Azimuth range for control beams in degrees.

ElevationRangeCtrl

float ***WinProp_Generate_Beams::ElevationRangeCtrl***
Elevation range for control beams in degrees.

MaxGainCtrl

float * ***WinProp_Generate_Beams::MaxGainCtrl***
Array of maximum gains for control beams in dBi. Needs to be an array of size NbrAzimuthBeamsCtrl * NbrElevationBeamsCtrl .

HorBeamwidthCtrl

float * ***WinProp_Generate_Beams::HorBeamwidthCtrl***
Array of 3-dB horizontal beamwidths for control beams in degrees. Needs to be an array of size NbrAzimuthBeamsCtrl * NbrElevationBeamsCtrl .

VerBeamwidthCtrl

float * ***WinProp_Generate_Beams::VerBeamwidthCtrl***
Array of 3-dB vertical beamwidths for control beams in degrees. Needs to be an array of size NbrAzimuthBeamsCtrl * NbrElevationBeamsCtrl .

NbrAzimuthBeamsData

int ***WinProp_Generate_Beams::NbrAzimuthBeamsData***
Number data beams in the azimuth domain

NbrElevationBeamsData

int ***WinProp_Generate_Beams::NbrElevationBeamsData***
Number data beams in the elevation domain

AzimuthRangeData

float ***WinProp_Generate_Beams::AzimuthRangeData***
Azimuth range for data beams in degrees.

ElevationRangeData

float ***WinProp_Generate_Beams::ElevationRangeData***
Elevation range for data beams in degrees.

MaxGainData

float * ***WinProp_Generate_Beams::MaxGainData***
Array of maximum gains for data beams in dBi. Needs to be an array of size NbrAzimuthBeamsData * NbrElevationBeamsData .

HorBeamwidthData

float * ***WinProp_Generate_Beams::HorBeamwidthData***
Array of 3-dB horizontal beamwidths for data beams in degrees. Needs to be an array of size NbrAzimuthBeamsData * NbrElevationBeamsData .

VerBeamwidthData

float * ***WinProp_Generate_Beams::VerBeamwidthData***
Array of 3-dB vertical beamwidths for data beams in degrees. Needs to be an array of size NbrAzimuthBeamsData * NbrElevationBeamsData .

Frequency

float ***WinProp_Generate_Beams::Frequency***
Frequency in MHz

PolAngle

float ***WinProp_Generate_Beams::PolAngle***
Linear Polarization angle in degrees

XPD

float ***WinProp_Generate_Beams::XPD***
Cross-polarization discrimination in dB

OutputFilename

WinProp_Filename ***WinProp_Generate_Beams::OutputFilename***
Name of output files without extension. For the file of the envelope and beam patterns, "_BeamPatterns" is appended to the filename. For the file of the envelope patterns, "_EnvelopePattern" is appended to the filename.

GenerateEnvelopeBeams

int ***WinProp_Generate_Beams::GenerateEnvelopeBeams***
Generate envelope patterns and a set of individual beams:

- 0 = No
- 1 = yes

GenerateEnvelope

int ***WinProp_Generate_Beams::GenerateEnvelope***
Generate only envelope in a separate file:

- 0 = No
- 1 = yes

The documentation was generated from the following file:

- `source/Beams/winprop_beams_types.h`

2.15.52 WinProp_Group_Dynamics

Data Fields

int
GroupID

WinProp_Dynamic_Sets
DynamicSets

Detailed Description

This structure is used to set the group dynamic sets defined in the *.nip file.

Field Documentation

GroupID

int ***WinProp_Group_Dynamics::GroupID***
Unique group index

DynamicSets

WinProp_Dynamic_Sets ***WinProp_Group_Dynamics::DynamicSets***
list with dynamic for group

The documentation was generated from the following file:

- `source/Interface/WP_Propagation.h`

2.15.53 WinProp_Legend

Data Fields

int
NrSamplingPoints

WinProp_SamplingPoint *
SamplingPoints

Detailed Description

Field Documentation

NrSamplingPoints

int ***WinProp_Legend::NrSamplingPoints***

SamplingPoints

WinProp_SamplingPoint * ***WinProp_Legend::SamplingPoints***

The documentation was generated from the following file:

- `source/Interface/Engine.h`

2.15.54 WinProp_Measurement

Data Fields

unsigned int
NrMeasurements

WinProp_Filename
MeasurementFile

WinProp_MeasurementPoint *
Points

Detailed Description

This structure is used to pass measurement data to the function OutdoorPlugIn_ComputePrediction in order to generate a calibration file (*.cal).

Field Documentation

NrMeasurements

unsigned int ***WinProp_Measurement::NrMeasurements***
Number of measurements

MeasurementFile

WinProp_Filename ***WinProp_Measurement::MeasurementFile***
Name of the measurement file

Points

WinProp_MeasurementPoint * ***WinProp_Measurement::Points***
Measurement points. See WinProp_MeasurementPoint

The documentation was generated from the following file:

- `source/Interface/WP_Measurements.h`

2.15.55 WinProp_MeasurementPoint

Data Fields

double
LocationX

double
LocationY

double
SignalLevel

double
PredictionLevel

int
ClutterClassID

int
StateLOS

unsigned int
PixelX

unsigned int
PixelY

Detailed Description

This structure is used to define a measurement point in the structure WinProp_Measurement.

Field Documentation

LocationX

double ***WinProp_MeasurementPoint::LocationX***
Location of measurement point (x coordinate).

LocationY

double ***WinProp_MeasurementPoint::LocationY***
Location of measurement point (y coordinate).

SignalLevel

double ***WinProp_MeasurementPoint::SignalLevel***
Measured signal level. The type (either power, path loss or field strength) depends on the computed result.

PredictionLevel

double ***WinProp_MeasurementPoint::PredictionLevel***
Predicted signal level. This is an output parameter.

ClutterClassID

int ***WinProp_MeasurementPoint::ClutterClassID***
Index of clutter class (if clutter classes are used). This is an output parameter.

StateLOS

int ***WinProp_MeasurementPoint::StateLOS***
Line of sight status. This is an output parameter.

- 1 = LOS
- 0 = NLOS

PixelX

unsigned int ***WinProp_MeasurementPoint::PixelX***
Location of pixel in result matrix (column). This is an output parameter.

PixelY

unsigned int ***WinProp_MeasurementPoint::PixelY***
Location of pixel in result matrix (row). This is an output parameter.

The documentation was generated from the following file:

- `source/Interface/WP_Measurements.h`

2.15.56 WinProp_Measurements

Data Fields

WinProp_Filename
CalibrationFile

int
NbrValues

*double **
CoordianteX

*double **
CoordianteY

*double **
Pathloss

Detailed Description

This structure is used pass measurement data to the computation function WinProp_Predict. Measurements are used to write a calibration file for the DPM (Dominant Path Model).

Field Documentation

CalibrationFile

WinProp_Filename ***WinProp_Measurements::CalibrationFile***
Name of calibration file (*.dpm) which will be used to store the information.

NbrValues

int ***WinProp_Measurements::NbrValues***
Number of measurement values, resp. number of elements in the arrays.

CoordianteX

double * ***WinProp_Measurements::CoordianteX***
x-coordinates of array elements

CoordianteY

double * ***WinProp_Measurements::CoordianteY***
y-coordinates of array elements

Pathloss

double * ***WinProp_Measurements::Pathloss***
Array with path loss values (dB)

The documentation was generated from the following file:

- `source/Interface/WP_Measurements.h`

2.15.57 WinProp_MonteCarloStatistics

Data Fields

float
nbrNotConnected

float
nbrBlocked

float
nbrServed

float
nbrNotAssigned

Detailed Description

Structure with statistical information of Monte Carlo analysis

Field Documentation

nbrNotConnected

float ***WinProp_MonteCarloStatistics::nbrNotConnected***
Number of not connected mobiles

nbrBlocked

float ***WinProp_MonteCarloStatistics::nbrBlocked***
Number of blocked mobiles

nbrServed

float ***WinProp_MonteCarloStatistics::nbrServed***
Number of served mobiles

nbrNotAssigned

float ***WinProp_MonteCarloStatistics::nbrNotAssigned***
Number of not assigned mobiles

The documentation was generated from the following file:

- `source/Interface/WP_Result.h`

2.15.58 WinProp_MS_AdditionalResults

Data Fields

char
outputPath [500]

int
filterOrder

int
enableASCIIoutput

int
channelMatricesPerPoint

int
channelMatricesPerRay

WINPROP_MS_CHANNEL_MATRIX_MODE
channelMatrixResultMode

int
conditionNumber

int
powerAzimuthSpectrum

double
powerAzimuthSpectrumResolution

int
propagationPaths

int
perStreamPower

int
perSubChannelPower

int
subChannelPowerSampleFreqStep

int
spreadLogLin

int
spreadPowerField

int
delaySpread

int
angularSpread

int
angularMean

int
dopplerShift

int
dopplerSpread

int
propSimF8

int
propSimAbsDelay

int
impedanceCouplingMatrix

int
channelCapacity

WINPROP_MS_CHANNEL_CAPACITY_MODE
capacityMode

float
capacityOutage

Detailed Description

Structure to specify additional outputs to be generated when using WinProp_SuperposeMS::compute. The overall power will be written as soon as this structure is specified.

Field Documentation

outputPath

char ***WinProp_MS_AdditionalResults::outputPath****[500]*
The path to which the additional results will be written

filterOrder

int ***WinProp_MS_AdditionalResults::filterOrder***
The filter order, set to zero to disable filtering of the results

enableASCIIoutput

int ***WinProp_MS_AdditionalResults::enableASCIIoutput***
Set to (1) to enable or (0) to disable output of ASCII files

channelMatricesPerPoint

int ***WinProp_MS_AdditionalResults::channelMatricesPerPoint***
The channel matrices per point (1) enabled or (0) disabled

channelMatricesPerRay

int ***WinProp_MS_AdditionalResults::channelMatricesPerRay***
The channel matrices per ray (1) enabled or (0) disabled

channelMatrixResultMode

WINPROP_MS_CHANNEL_MATRIX_MODE
WinProp_MS_AdditionalResults::channelMatrixResultMode
The channel matrix result mode

conditionNumber

int ***WinProp_MS_AdditionalResults::conditionNumber***
The condition number (1) enabled or (0) disabled

powerAzimuthSpectrum

int ***WinProp_MS_AdditionalResults::powerAzimuthSpectrum***
The power azimuth spectrum (1) enabled or (0) disabled

powerAzimuthSpectrumResolution

double ***WinProp_MS_AdditionalResults::powerAzimuthSpectrumResolution***
The angle resolution for the power azimuth spectrum resolution in degree (value between [0.25, 180.0])

propagationPaths

int ***WinProp_MS_AdditionalResults::propagationPaths***
The propagation paths (1) enabled or (0) disabled

perStreamPower

int ***WinProp_MS_AdditionalResults::perStreamPower***
The per stream power (1) enabled or (0) disabled

perSubChannelPower

int ***WinProp_MS_AdditionalResults::perSubChannelPower***
The per sub channel power (1) enabled or (0) disabled

subChannelPowerSampleFreqStep

int ***WinProp_MS_AdditionalResults::subChannelPowerSampleFreqStep***
The sub channel power sample frequency step

spreadLogLin

int ***WinProp_MS_AdditionalResults::spreadLogLin***
Compute delay/angular spread/mean based on (1) linear or (0) logarithmic values

spreadPowerField

int ***WinProp_MS_AdditionalResults::spreadPowerField***
Compute delay/angular spread/mean based on (1) fieldstrength or (0) power values

delaySpread

int ***WinProp_MS_AdditionalResults::delaySpread***
The delay spread (1) enabled or (0) disabled

angularSpread

int ***WinProp_MS_AdditionalResults::angularSpread***
The angular spread (1) enabled or (0) disabled

angularMean

int ***WinProp_MS_AdditionalResults::angularMean***
The angular mean (1) enabled or (0) disabled

dopplerShift

int ***WinProp_MS_AdditionalResults::dopplerShift***
The doppler shift (1) enabled or (0) disabled, only available for trajectories

dopplerSpread

int ***WinProp_MS_AdditionalResults::dopplerSpread***
The doppler spread (1) enabled or (0) disabled, only available for trajectories

propSimF8

int ***WinProp_MS_AdditionalResults::propSimF8***
The PropSimF8 input (1) enabled or (0) disabled, only available for trajectories

propSimAbsDelay

int ***WinProp_MS_AdditionalResults::propSimAbsDelay***
The PropSim results with (1) absolute or (0) relative delays, only available for trajectories

impedanceCouplingMatrix

int ***WinProp_MS_AdditionalResults::impedanceCouplingMatrix***
The impedance coupling matrix (1) enabled or (0) disabled

channelCapacity

int ***WinProp_MS_AdditionalResults::channelCapacity***
The channel capacity (1) enabled or (0) disabled

capacityMode

WINPROP_MS_CHANNEL_CAPACITY_MODE ***WinProp_MS_AdditionalResults::capacityMode***
The channel capacity mode

capacityOutage

float ***WinProp_MS_AdditionalResults::capacityOutage***
Outage probability for channel capacity in percent

The documentation was generated from the following file:

- `source/SuperposeMS/winprop_superposems_types.h`

2.15.59 WinProp_MS_Para

Data Fields

int
fullPolarimetricComputation

int
coherentSuperposition

int
nrRays

WINPROP_MS_CHANNEL_TYPE
channelType

float
channelBandwidth

WINPROP_MS_NORMALIZE_TYPE
channelNormalization

int
normalizeOnlyForCapacityPropSim

WINPROP_MS_SNIR_TYPE
snirMode

float
SNIRmean

float
SNIRAdditionalInterference

float
AntennaDynamicLimitBS

float
AntennaDynamicLimitMS

Detailed Description

Structure to specify the computation parameters of a WinProp_SuperposeMS computation.

Field Documentation

fullPolarimetricComputation

int ***WinProp_MS_Para::fullPolarimetricComputation***
Full polarimetric computation (1) enabled or (0) disabled

coherentSuperposition

int ***WinProp_MS_Para::coherentSuperposition***
Coherent superposition (1) enabled or (0) disabled

nrRays

int ***WinProp_MS_Para::nrRays***
Maximum number of rays which are considered from the propagation per pixel (should be power of 2)

channelType

WINPROP_MS_CHANNEL_TYPE ***WinProp_MS_Para::channelType***
Type of the channel

channelBandwidth

float ***WinProp_MS_Para::channelBandwidth***
The channel bandwidth in (Hz)

channelNormalization

WINPROP_MS_NORMALIZE_TYPE ***WinProp_MS_Para::channelNormalization***
The channel normalization

normalizeOnlyForCapacityPropSim

int ***WinProp_MS_Para::normalizeOnlyForCapacityPropSim***
Normalize only for channel capacity and PropSim F8 output (1) or for all computations (0)

snirMode

WINPROP_MS_SNIR_TYPE ***WinProp_MS_Para::snirMode***
The snir mode

SNIRmean

float ***WinProp_MS_Para::SNIRmean***
The SNIR mean (dB), only relevant in case snirMode is set to WINPROP_MS_SNIR_MEAN

SNIRAdditionalInterference

float ***WinProp_MS_Para::SNIRAdditionalInterference***
The additional SNIR interference, only relevant in case snirMode is set to WINPROP_MS_SNIR_CALC

AntennaDynamicLimitBS

float ***WinProp_MS_Para::AntennaDynamicLimitBS***
The dynamic limit of transmitting antennas (dB)

AntennaDynamicLimitMS

float ***WinProp_MS_Para::AntennaDynamicLimitMS***
The dynamic limit of receiving antennas (dB)

The documentation was generated from the following file:

- `source/SuperposeMS/winprop_superposems_types.h`

2.15.60 WinProp_ParaHybrid

Data Fields

int
MaterialsIndividual

int
WithSubdivisions

int
Model

double
Resolution

int
NrHeights

*double **
Heights

int
HeightMode

int
FloorSelection

int
DiffractionModel

int
IRTInteractionsMaxTotal

int
IRTInteractionsMaxReflections

int
IRTInteractionsMaxDiffractions

int
IRTInteractionsMaxTransmissions

float
TransmissionLoss

float
ReflectionLoss

float
DiffractionLossInMin

float
DiffractionLossInMax

float
DiffractionLossOut

float
Epsilon

float
Mu

float
Sigma

Detailed Description

This structure is used for Hybrid scenarios (e.g., CNP mode). This is not yet supported.

Field Documentation

MaterialsIndividual

int ***WinProp_ParaHybrid::MaterialsIndividual***
Individual materials or not (only in combination with WinProp database)

- 0 = default materials
- 1 = individual (odb, opb)

WithSubdivisions

int ***WinProp_ParaHybrid::WithSubdivisions***
Parameter to enable the consideration of subdivisions

- 1 = subdivisions considered
- 0 = subdivisions not considered

Model

int ***WinProp_ParaHybrid::Model***
This parameter should be set to 0 (Indoor model)

Resolution

double ***WinProp_ParaHybrid::Resolution***
Indoor resolution (meters)

NrHeights

int ***WinProp_ParaHybrid::NrHeights***
Number of prediction heights of the indoor database in the CNP scenario.

Heights

*double ** ***WinProp_ParaHybrid::Heights***
Array of the prediction heights

HeightMode

int ***WinProp_ParaHybrid::HeightMode***
Heights relative to floors or relative to ground.

- 0 = Heights relative to ground
- 1 = Heights relative to floors
- 2 = Heights relative to roofs
- 3 = Heights relative to lowest roof
- 4 = Heights relative to highest roof
- 5 = Heights relative to urban height

FloorSelection

int ***WinProp_ParaHybrid::FloorSelection***
Parameter to select results from which floor to copy to the urban results

- -1 = Do not copy CNP results to urban results
- 0 = Copy lowest floor to urban results
- 1 = Copy highest floor to urban results

DiffractionModel

int ***WinProp_ParaHybrid::DiffractionModel***
Diffraction mode for Indoor IRT in CNP mode.

- 0 = Fresnel coefficients for reflection and transmission, and GTD/UTD for diffraction
- 1 = Empirical losses for reflection, transmission and diffraction.

IRTInteractionsMaxTotal

int ***WinProp_ParaHybrid::IRTInteractionsMaxTotal***
Max. number of interactions in one ray for indoor IRT in CNP mode.

IRTInteractionsMaxReflections

int ***WinProp_ParaHybrid::IRTInteractionsMaxReflections***
Max. number of reflections in one ray for indoor IRT in CNP mode.

IRTInteractionsMaxDiffractions

int ***WinProp_ParaHybrid::IRTInteractionsMaxDiffractions***
Max. number of diffractions in one ray for indoor IRT in CNP mode.

IRTInteractionsMaxTransmissions

int ***WinProp_ParaHybrid::IRTInteractionsMaxTransmissions***
Max. number of transmissions in one ray for indoor IRT in CNP mode.

TransmissionLoss

float ***WinProp_ParaHybrid::TransmissionLoss***
Transmission loss (dB) of the default material for empirical interaction model. Only relevant if MaterialsIndividual = default materials

ReflectionLoss

float ***WinProp_ParaHybrid::ReflectionLoss***
Reflection loss (dB) of the default material for empirical interaction model. Only relevant if MaterialsIndividual = default materials

DiffractionLossInMin

float ***WinProp_ParaHybrid::DiffractionLossInMin***
Min. diffraction loss (dB) of incident rays of the default material for empirical interaction model. Only relevant if MaterialsIndividual = default materials

DiffractionLossInMax

float ***WinProp_ParaHybrid::DiffractionLossInMax***
Max. diffraction loss (dB) of incident rays of the default material for empirical interaction model. Only relevant if MaterialsIndividual = default materials

DiffractionLossOut

float ***WinProp_ParaHybrid::DiffractionLossOut***
Diffraction loss (dB) of diffracted rays of the default material for empirical interaction model. Only relevant if MaterialsIndividual = default materials

Epsilon

float ***WinProp_ParaHybrid::Epsilon***
Dielectricity (relative value) of the default material for Fresnel and GTD/UTD interaction model. Only relevant if MaterialsIndividual = default materials

Mu

float ***WinProp_ParaHybrid::Mu***
Permeability (relative value) of the default material for Fresnel and GTD/UTD interaction model. Only relevant if MaterialsIndividual = default materials

Sigma

float ***WinProp_ParaHybrid::Sigma***
Conductivity (S/m) of the default material for Fresnel and GTD/UTD interaction model. Only relevant if MaterialsIndividual = default materials

The documentation was generated from the following file:

- `source/Interface/WP_ParaUrban.h`

2.15.61 WinProp_ParaMain

Data Fields

WINPROP_OUTDOOR_SCENARIO_MODE
ScenarioMode

PREDMODEL_URBAN
PredictionModelUrban

PREDMODEL_RURAL
PredictionModelRural

unsigned int
PredictionLargeArea

double
UrbanLowerLeftX

double
UrbanLowerLeftY

double
UrbanUpperRightX

double
UrbanUpperRightY

double
RuralLowerLeftX

double
RuralLowerLeftY

double
RuralUpperRightX

double
RuralUpperRightY

double
Resolution

int
NrLayers

*double **
PredictionHeights

int
NrTimeInstances

*double **
TimeInstances

int
AbsolutePredictionHeight

double
VectorRadius

WINPROP_BREAKPOINT_MODE
BreakpointMode

double
BreakpointFactor

double
BreakpointOffset

double
BreakpointDistance

WINPROP_BUILDINGS_MODE
BuildingsMode

double
BuildingHeights

int
BuildingHeightsAbs

int
BuildingsMaterialsInd

double
BuildingsTransmissionLoss

URBAN_BUILDINGS
Buildings

WinProp_Filename
BuildingsName

WINPROP_INDOOR_COVERAGE_MODE
    BuildingsIndoorCoverage

WINPROP_OUTDOOR_TOPO_MODE
    TopographyMode

TOPOGRAPHY
    Topography

WinProp_Filename
    TopographyName

bool
    TopographyIndexDatabase

int
    TopographyHeightMode

WINPROP_CLUTTER_MODE
    ClutterMode

WINPROP_CLUTTER_HEIGHT_MODE
    ClutterConsideration

WINPROP_CLUTTER_ATTENUATION_MODE
    ClutterAttenuation

int
    ClutterIntervalsNumber

float
    ClutterIntervalsLength

int
    ClutterCalibration

CLUTTER
    Clutter

WinProp_Filename
    ClutterName

bool
    ClutterIndexDatabase

WinProp_Filename
    ClutterClassesName

WINPROP_VEGETATION_MODE
    VegetationMode

WinProp_Filename
    VegetationName

int
    VegetationPropertiesInd

double
VegetationLossFixed

double
VegetationLossPerMeter

double
VegetationResolution

unsigned int
BuildingsPixelMode

TOPOGRAPHY
BuildingsPixel

WinProp_Filename
BuildingsPixelName

int
BuildingsPixelHeightMode

double
BuildingsPixelHeightTolerance

int
ConsiderDatabasePolygons

int
NrDatabasePolygons

WinProp_Polygon *
DatabasePolygons

WinProp_Additional_Freq_Loss
FreqDepLosses

WinProp_Propagation_Results *
OutputResults

double
Offset

unsigned int
DistributedPrediction

unsigned int
Filter

unsigned int
FilterOrder

unsigned int
FilterMode

bool
UseAntennaLimit

double
AntennaLimit

WinProp_ErrorString
ErrorMessageMain

WinProp_ErrorString
ErrorMessageSub

int
SupressMessageBoxes

int
ModeAPI

int
MaxPathLossEnabled

float
MaxPathLoss

int
MaxDynamicPathsEnabled

float
MaxDynamicPaths

int
MaxNumberPathsUsed

int
MaxNumberPaths

int
RaysLogLin

int
RaysPowerField

int
IndoorPredictionOnly

double
ValueNotComputed

double
BuildingsIndoorDecrease

int
MultiThreading

int
FullPolarimetric

int
RuralModeCNP

int
PredictionSurfaces

Detailed Description

WinProp_ParaMain: Structure for urban parameters for outdoor plug-in. This structure is used to define most of the parameters for a prediction computation accomplished with the OutdoorPlugIn_ComputePrediction function.

Field Documentation

ScenarioMode

WINPROP_OUTDOOR_SCENARIO_MODE ***WinProp_ParaMain::ScenarioMode***
The scenario mode.

PredictionModelUrban

PREDMODEL_URBAN ***WinProp_ParaMain::PredictionModelUrban***
Selection of urban prediction model see PREDMODEL_URBAN. Only relevant for SCENARIOMODE_URBAN. See ScenarioMode.

PredictionModelRural

PREDMODEL_RURAL ***WinProp_ParaMain::PredictionModelRural***
Selection of rural prediction model see PREDMODEL_RURAL. Only relevant for SCENARIOMODE_RURAL. See ScenarioMode.

PredictionLargeArea

unsigned int ***WinProp_ParaMain::PredictionLargeArea***
Use large area mode (multi scenario)

UrbanLowerLeftX

double ***WinProp_ParaMain::UrbanLowerLeftX***
Lower left corner of urban prediction area (x coordinate).

UrbanLowerLeftY

double ***WinProp_ParaMain::UrbanLowerLeftY***
Lower left corner of urban prediction area (y coordinate).

UrbanUpperRightX

double ***WinProp_ParaMain::UrbanUpperRightX***
Upper right corner of urban prediction area (x coordinate).

UrbanUpperRightY

double ***WinProp_ParaMain::UrbanUpperRightY***
Upper right corner of urban prediction area (y coordinate).

RuralLowerLeftX

double ***WinProp_ParaMain::RuralLowerLeftX***
Lower left corner of rural prediction area (x coordinate).

RuralLowerLeftY

double ***WinProp_ParaMain::RuralLowerLeftY***
Lower left corner of rural prediction area (y coordinate).

RuralUpperRightX

double ***WinProp_ParaMain::RuralUpperRightX***
Upper right corner of rural prediction area (x coordinate).

RuralUpperRightY

double ***WinProp_ParaMain::RuralUpperRightY***
Upper right corner of rural prediction area (y coordinate).

Resolution

double ***WinProp_ParaMain::Resolution***
Resolution for prediction (meter). In case of prediction model 3D Intelligent Ray Tracing (PredictionModelUrban set to PREDMODEL_IRT) only preprocessed resolutions can be used.

NrLayers

int ***WinProp_ParaMain::NrLayers***
Number of prediction layers

PredictionHeights

double * ***WinProp_ParaMain::PredictionHeights***
Heights for prediction (meter). Array of length NrLayers. In case of prediction model 3D Intelligent Ray Tracing (PredictionModelUrban set to PREDMODEL_IRT) only preprocessed heights can be used.

NrTimeInstances

int ***WinProp_ParaMain::NrTimeInstances***
Number of time instances for time-variant simulation.

TimeInstances

double * ***WinProp_ParaMain::TimeInstances***
Time instances (second) for time-variant simulation. Array of length NrTimeInstances.

AbsolutePredictionHeight

int ***WinProp_ParaMain::AbsolutePredictionHeight***
Consider prediction height as absolute above sea level. Supported only by some prediction models.

- 0 = prediction height is relative

- 1 = prediction height is absolute

VectorRadius

double ***WinProp_ParaMain::VectorRadius***
Vector radius if necessary. Not yet used

BreakpointMode

WINPROP_BREAKPOINT_MODE ***WinProp_ParaMain::BreakpointMode***
Computation of breakpoint distance.

BreakpointFactor

double ***WinProp_ParaMain::BreakpointFactor***
Factor for determination of breakpoint distance, allowed range [1, 20]. Only required if BreakpointMode is NOT set to BREAKPOINT_FIXED.

BreakpointOffset

double ***WinProp_ParaMain::BreakpointOffset***
Offset (meter) for determination of breakpoint distance, allowed range [0, 10000]. Only required if BreakpointMode is NOT set to BREAKPOINT_FIXED.

BreakpointDistance

double ***WinProp_ParaMain::BreakpointDistance***
Fixed distance (meter) for breakpoint, allowed range [0, 10000]. Only required if BreakpointMode is set to BREAKPOINT_FIXED.

BuildingsMode

WINPROP_BUILDINGS_MODE ***WinProp_ParaMain::BuildingsMode***
Type of the building database.

BuildingHeights

double ***WinProp_ParaMain::BuildingHeights***
Default building heights

BuildingHeightsAbs

int ***WinProp_ParaMain::BuildingHeightsAbs***
Mode for building heights. Only relevant if BuildingsMode is set to BUILDINGSMODE_RAM.

- 0 = height relative to ground
- 1 = height absolute

BuildingsMaterialsInd

int ***WinProp_ParaMain::BuildingsMaterialsInd***
Individual materials or not. Only applicable in combination with WinProp database (BuildingsMode set to BUILDINGSMODE_BINARY, BUILDINGSMODE_IRT or BUILDINGSMODE_UDP (.odb, .oib, .opb))

- 0 = default materials

- 1 = individual materials

BuildingsTransmissionLoss

double ***WinProp_ParaMain::BuildingsTransmissionLoss***
Transmission loss of buildings (dB)

Buildings

URBAN_BUILDINGS ***WinProp_ParaMain::Buildings***
Data structure if the buildings should be read from RAM (BuildingsMode set to BUILDINGSMODE_RAM)

BuildingsName

WinProp_Filename ***WinProp_ParaMain::BuildingsName***
Filename of WinProp vector database. Can be relative to project file. Only applicable if BuildingsMode is not set to BUILDINGSMODE_RAM or BUILDINGSMODE_NONE.

BuildingsIndoorCoverage

WINPROP_INDOOR_COVERAGE_MODE ***WinProp_ParaMain::BuildingsIndoorCoverage***
Prediction of indoor coverage (inside building polygons).

TopographyMode

WINPROP_OUTDOOR_TOPO_MODE ***WinProp_ParaMain::TopographyMode***
The topography mode.

Topography

TOPOGRAPHY ***WinProp_ParaMain::Topography***
Topography data, only relevant in case TopographyMode is set to TOPOMODE_RAM

TopographyName

WinProp_Filename ***WinProp_ParaMain::TopographyName***
Filename of the topography database, only relevant in case TopographyMode is set to TOPOMODE_SEPARATE

TopographyIndexDatabase

bool ***WinProp_ParaMain::TopographyIndexDatabase***
True to topography index database in TopographyName, only relevant in case TopographyMode is set to TOPOMODE_SEPARATE

TopographyHeightMode

int ***WinProp_ParaMain::TopographyHeightMode***
Topo database includes buildings (0) or not (1)

ClutterMode

WINPROP_CLUTTER_MODE ***WinProp_ParaMain::ClutterMode***
The clutter mode.

ClutterConsideration

WINPROP_CLUTTER_HEIGHT_MODE ***WinProp_ParaMain::ClutterConsideration***
The clutter heights type.

ClutterAttenuation

WINPROP_CLUTTER_ATTENUATION_MODE ***WinProp_ParaMain::ClutterAttenuation***
Mode for clutter frequency depending losses.

ClutterIntervalsNumber

int ***WinProp_ParaMain::ClutterIntervalsNumber***
Clutter attenuation; Number of intervals with different weight factor. Only relevant if ClutterAttenuation is set to CLUTTERATTENUATION_RAY_DECREASE

ClutterIntervalsLength

float ***WinProp_ParaMain::ClutterIntervalsLength***
Clutter attenuation; Min length of interval. Only relevant if ClutterAttenuation is set to CLUTTERATTENUATION_RAY_DECREASE

ClutterCalibration

int ***WinProp_ParaMain::ClutterCalibration***
Calibrate clutter classes.

- 0 = no
- 1 = yes, without calculating mean error
- 2 = yes, with mean error as an output

Clutter

CLUTTER ***WinProp_ParaMain::Clutter***
Clutter data, only relevant in case ClutterMode is set to CLUTTERMODE_RAM

ClutterName

WinProp_Filename ***WinProp_ParaMain::ClutterName***
Filename of the clutter database, only relevant in case ClutterMode is set to CLUTTERMODE_FILE

ClutterIndexDatabase

bool ***WinProp_ParaMain::ClutterIndexDatabase***
True to clutter index database in ClutterName, only relevant in case ClutterMode is set to CLUTTERMODE_FILE

ClutterClassesName

WinProp_Filename ***WinProp_ParaMain::ClutterClassesName***
Filename of the clutter classes file, only relevant in case ClutterMode is set to CLUTTERMODE_FILE

VegetationMode

WINPROP_VEGETATION_MODE ***WinProp_ParaMain::VegetationMode***
The vegetation mode.

VegetationName

WinProp_Filename ***WinProp_ParaMain::VegetationName***
Filename of the vegetation file, only relevant in case VegetationMode is set to either VEGETATION_PIXEL or VEGETATION_SEPARATE. Filename without extension

VegetationPropertiesInd

int ***WinProp_ParaMain::VegetationPropertiesInd***
Individual vegetation properties.

- 0 = no
- 1 = yes

VegetationLossFixed

double ***WinProp_ParaMain::VegetationLossFixed***
The vegetation loss fixed

VegetationLossPerMeter

double ***WinProp_ParaMain::VegetationLossPerMeter***
The vegetation loss per meter

VegetationResolution

double ***WinProp_ParaMain::VegetationResolution***
The resolution used to create vegetation blocks out of large vegetation polygons in case of topography. This is done to allow the vegetation to adapt to the topography. Individual vegetation objects have at max the size of the set number in square meter. All created blocks together will have the same shape as the original polygons.

BuildingsPixelMode

unsigned int ***WinProp_ParaMain::BuildingsPixelMode***
Use urban buildings from pixel database.

- 0 = no buildings from pixel database
- 1 = use buildings from RAM See also BuildingsPixel
- 2 = load buildings from pixel database, file has to be specified in BuildingsPixelName

BuildingsPixel

TOPOGRAPHY ***WinProp_ParaMain::BuildingsPixel***
Structure for pixel buildings

BuildingsPixelName

WinProp_Filename ***WinProp_ParaMain::BuildingsPixelName***
Filename of the pixel buildings file, only relevant in case BuildingsPixelMode is set to 2.

BuildingsPixelHeightMode

int ***WinProp_ParaMain::BuildingsPixelHeightMode***
Mode for pixel building heights.

- 0 = height relative to ground
- 1 = height absolute

BuildingsPixelHeightTolerance

double ***WinProp_ParaMain::BuildingsPixelHeightTolerance***
Height tolerance [m] for pixel building heights

ConsiderDatabasePolygons

int ***WinProp_ParaMain::ConsiderDatabasePolygons***
Consider the database polygons if they are defined in the scenario (enabled/disabled)

- 0 = don't consider database polygons
- 1 = consider database polygons

NrDatabasePolygons

int ***WinProp_ParaMain::NrDatabasePolygons***
Number of database polygons in the scenario.

DatabasePolygons

WinProp_Polygon * ***WinProp_ParaMain::DatabasePolygons***
The database polygons in the scenario.

FreqDepLosses

WinProp_Additional_Freq_Loss ***WinProp_ParaMain::FreqDepLosses***
This structure allows to define additional frequency dependent losses (atmospheric losses).

OutputResults

WinProp_Propagation_Results * ***WinProp_ParaMain::OutputResults***
This structure allows to define additional results to be computed and written in binary WinProp format. Disabled by setting to NULL.

Offset

double ***WinProp_ParaMain::Offset***
Offset (dB) applied to all receiver pixels, allowed range [-100, 100].

DistributedPrediction

unsigned int ***WinProp_ParaMain::DistributedPrediction***
Distributed computations. Only modifies the output of errors, does not distribute the computations automatically across multiple PCs.

- 0 if prediction on local PC
- 1 if remote/distributed prediction

Filter

unsigned int ***WinProp_ParaMain::Filter***
Filter the results.

- 0 = no usage of filters
- 1 = use filters

FilterOrder

unsigned int ***WinProp_ParaMain::FilterOrder***
The filter order, if filter is used. The order must be an odd number, e.g. 1, 3, 5, ..See also Filter

FilterMode

unsigned int ***WinProp_ParaMain::FilterMode***
Type of filter, if filter is usedSee also FilterNOT USED ANYMORE

- FILTERMODE_MEAN : Mean value filtering.
- FILTERMODE_MEDIAN : Median value filtering.

UseAntennaLimit

bool ***WinProp_ParaMain::UseAntennaLimit***
True to limit the dynamic of the antennas

AntennaLimit

double ***WinProp_ParaMain::AntennaLimit***
The maximum dynamic of antennas. Only applicable if UseAntennaLimit is set to true

ErrorMessageMain

WinProp_ErrorString ***WinProp_ParaMain::ErrorMessageMain***
This is an output parameter! Contains an error message if an error occurs during a prediction.

ErrorMessageSub

WinProp_ErrorString ***WinProp_ParaMain::ErrorMessageSub***
This is an output parameter! Contains further information of the error in ErrorMessageMain, if an error occurs during a prediction.

SupressMessageBoxes

int ***WinProp_ParaMain::SupressMessageBoxes***
Suppress error messages.

- 0 = don't suppress errors
- 1 = suppress errors

ModeAPI

int ***WinProp_ParaMain::ModeAPI***
For internal usage only. DO NOT CHANGE!!

MaxPathLossEnabled

int ***WinProp_ParaMain::MaxPathLossEnabled***
Max absolute value of path loss of rays used to compute prediction results (enabled/disabled)

- 0 = don't limit max absolute value of path loss per ray
- 1 = limit max absolute value of path loss per ray, maximum has to be specified in MaxPathLoss

MaxPathLoss

float ***WinProp_ParaMain::MaxPathLoss***
The max absolute value of path loss of rays. Only applicable if MaxPathLossEnabled is set to 1

MaxDynamicPathsEnabled

int ***WinProp_ParaMain::MaxDynamicPathsEnabled***
Max dynamic range of path loss of rays used to compute prediction results (enabled/disabled)

- 0 = don't limit the dynamic range of rays
- 1 = limit the dynamic range of rays, maximum has to be specified in MaxDynamicPaths

MaxDynamicPaths

float ***WinProp_ParaMain::MaxDynamicPaths***
The maximum dynamic range path loss of rays. Only applicable if MaxDynamicPathsEnabled is set to 1

MaxNumberPathsUsed

int ***WinProp_ParaMain::MaxNumberPathsUsed***
Limit the number of rays which are computed.

- 0 = don't limit the number of rays
- 1 = limit the number of rays, maximum has to be specified in MaxNumberPaths

MaxNumberPaths

int ***WinProp_ParaMain::MaxNumberPaths***
The maximum number rays. Only applicable if MaxNumberPathsUsed is set to 1

RaysLogLin

int ***WinProp_ParaMain::RaysLogLin***
Define if the delay spread and angular spreads are computed

- 0 = with logarithmic values
- 1 = with linear values

RaysPowerField

int ***WinProp_ParaMain::RaysPowerField***
Define if the delay spread and angular spreads are computed based on

- 0 = power values
- 1 = field strength values

IndoorPredictionOnly

int ***WinProp_ParaMain::IndoorPredictionOnly***
Return results for indoor locations only from all computed results.

- 0 = get indoor and outdoor results
- 1 = get only indoor results

ValueNotComputed

double ***WinProp_ParaMain::ValueNotComputed***
The value, which represents not computed results

BuildingsIndoorDecrease

double ***WinProp_ParaMain::BuildingsIndoorDecrease***
For exponential decrease for variable decrease model, allowed range [0, 10]. Only relevant if BuildingsIndoorCoverage is set to COVERAGEINDOOR_USERDEFINED

MultiThreading

int ***WinProp_ParaMain::MultiThreading***
Number of threads being used for the computation of the particular antenna. Needs to be greater than zero. Currently not supported by any urban prediction model.

FullPolarimetric

int ***WinProp_ParaMain::FullPolarimetric***
Full polarimetric simulation (enabled/disabled)

- 0 = standard polarimetric analysis with total gain pattern only
- 1 = full polarimetric analysis with .ffe file or two gains patterns, one for vertical and one for horizontal polarisation

RuralModeCNP

int ***WinProp_ParaMain::RuralModeCNP***
RuralModeCNP (enabled/disabled)

- 0 = disabled
- 1 = enabled

PredictionSurfaces

int ***WinProp_ParaMain::PredictionSurfaces***
Enable prediction on building surfaces defined in the database

- 0 = Disable prediction on building surfaces
- 1 = Enable prediction on building surfaces

The documentation was generated from the following file:

- `source/Interface/WP_ParaUrban.h`

2.15.62 WinProp_ParaRural

Data Fields

WINPROP_KE_MODEL
KnifeEdge

WINPROP_HATA_SUB_MODEL
HataSubMode

double
ExponentLOS

double
ExponentLOSaB

int
Scat3D_Enabled

int
Scat3D_GroundProperties

double
Scat3D_RelativePermittivity

double
Scat3D_Conductivity

double
Scat3D_StdDivHeights

double
Scat3D_CorrelationLength

int
Scat3D_NrScatteringElements

int
Scat3D_MaxNrScatteringElements

int
Scat3D_PreprocessingMode

WinProp_Filename
Scat3D_PreproFile

int
UseExactCoordinates

int
HataFrequencyMode

int
DiffractionMaxNumber

double
CurvatureRadiusFactor

int
ConsiderEarthCurvature

Model_DetTwoRay
DetTwoRayModelParas

Detailed Description

This structure is used for the definition of rural prediction parameters. The structure is additionally passed to the OutdoorPlugIn_ComputePrediction function if a rural prediction should be computed.

Field Documentation

KnifeEdge

WINPROP_KE_MODEL ***WinProp_ParaRural::KnifeEdge***
Knife edge model.

HataSubMode

WINPROP_HATA_SUB_MODEL ***WinProp_ParaRural::HataSubMode***
Sub model of Hata.

ExponentLOS

double ***WinProp_ParaRural::ExponentLOS***
Path loss exponent for LOS areas, allowed range [1, 10]

ExponentLOSaB

double ***WinProp_ParaRural::ExponentLOSaB***
Path loss exponent for NLOS areas, allowed range [1, 10]

Scat3D_Enabled

int ***WinProp_ParaRural::Scat3D_Enabled***
Boolean to enable/disable computation of 3D Scattering (NOT YET SUPPORTED)

- 0 = disabled
- 1 = enabled

Scat3D_GroundProperties

int ***WinProp_ParaRural::Scat3D_GroundProperties***
Boolean to enable/disable consideration of ground properties

- 0 = disabled
- 1 = enabled

Scat3D_RelativePermittivity

double ***WinProp_ParaRural::Scat3D_RelativePermittivity***
Relative permittivity of medium

Scat3D_Conductivity

double ***WinProp_ParaRural::Scat3D_Conductivity***
Conductivity of medium

Scat3D_StdDivHeights

double ***WinProp_ParaRural::Scat3D_StdDivHeights***

Scat3D_CorrelationLength

double ***WinProp_ParaRural::Scat3D_CorrelationLength***

Scat3D_NrScatteringElements

int ***WinProp_ParaRural::Scat3D_NrScatteringElements***

Scat3D_MaxNrScatteringElements

int ***WinProp_ParaRural::Scat3D_MaxNrScatteringElements***

Scat3D_PreprocessingMode

int ***WinProp_ParaRural::Scat3D_PreprocessingMode***

Scat3D_PreproFile

WinProp_Filename ***WinProp_ParaRural::Scat3D_PreproFile***

UseExactCoordinates

int ***WinProp_ParaRural::UseExactCoordinates***
Select the usage of exact coordinates or an approximation

- 1 = Exact coordinates
- 0 = Approximation

HataFrequencyMode

int ***WinProp_ParaRural::HataFrequencyMode***
Select the usage of Hata or Extended Hata model

- 0 = Hata model (150 MHz to 1500 MHz)
- 1 = Extended Hata model (30 MHz to 3000 MHz)

DiffractionMaxNumber

int ***WinProp_ParaRural::DiffractionMaxNumber***
Maximum number of diffractions if Knife Edge model is used

CurvatureRadiusFactor

double ***WinProp_ParaRural::CurvatureRadiusFactor***
Factor for the Radius of the earth

ConsiderEarthCurvature

int ***WinProp_ParaRural::ConsiderEarthCurvature***
Parameter to consider earth curvature

- 0 = radius not considered
- 1 = radius considered

DetTwoRayModelParas

Model_DetTwoRay ***WinProp_ParaRural::DetTwoRayModelParas***
The parameter for the deterministic two ray model

The documentation was generated from the following file:

- `source/Interface/WP_ParaRural.h`

2.15.63 WinProp_Pattern

Data Fields

WINPROP_PATTERN_MODE
Mode

WinProp_Filename
Filename

WinProp_Filename
CompoFilename

int
NrVertical

*double **
AnglesVertical

double *
GainVertical

int
NrHorizontal

double *
AnglesHorizontal

double *
GainHorizontal

double **
Gain3D

double **
Phase3D

WINPROP_PATTERN_RESOLUTION
Resolution3D

WinProp_Antenna_Array *
AntennaArray

Detailed Description

This structure is used for the definition of the antenna pattern. It is used in WinProp_Antenna. Information: If "Filename" is not NULL, this pattern is loaded and used. If "Filename" is NULL and "NrVertical" and "NrHorizontal" is not 0, the values from the vertical and horizontal patterns are used to determine the 3D pattern. If "NrVertical" and "NrHorizontal" are 0, the values of the 3D pattern in "Gain3D" and "Phase3D" are considered. This means the following priority: Load from file => 2x2D pattern => 3D pattern.

Orientation of Antenna Pattern Orientation of pattern is as follows.

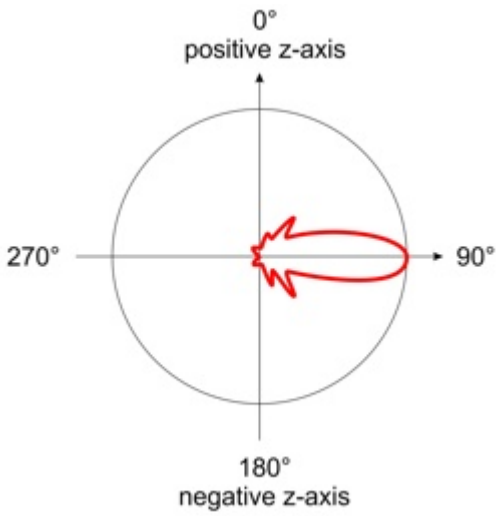

Figure 6: Vertical pattern

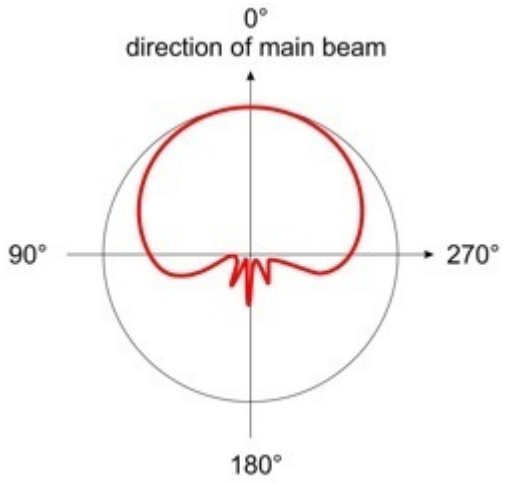

Figure 7: Horizontal pattern

Field Documentation

Mode

WINPROP_PATTERN_MODE ***WinProp_Pattern::Mode***
Mode for data in pattern.

Filename

WinProp_Filename ***WinProp_Pattern::Filename***
Filename of the supported antenna pattern, if it should be read from file. If this is NULL, the antenna pattern data is available in the following variables.

CompoFilename

WinProp_Filename ***WinProp_Pattern::CompoFilename***
Filename of the supported antenna pattern contained in the component database

NrVertical

int ***WinProp_Pattern::NrVertical***
Number of points in vertical direction.

AnglesVertical

double * ***WinProp_Pattern::AnglesVertical***
Array with angles for vertical pattern. The orientation of the antenna is described in the figure below.

GainVertical

double * ***WinProp_Pattern::GainVertical***
Array with gain values (dBi) for vertical pattern.

NrHorizontal

int ***WinProp_Pattern::NrHorizontal***
Number of points in horizontal direction.

AnglesHorizontal

double * ***WinProp_Pattern::AnglesHorizontal***
Array with angles for horizontal pattern. The orientation of the antenna is described in the figure below.

GainHorizontal

double * ***WinProp_Pattern::GainHorizontal***
Array with gain values (dBi) for horizontal pattern.

Gain3D

double ** ***WinProp_Pattern::Gain3D***
Array with gain values (dBi) for 3D pattern. This array must have the exact size of 180 vertical elements and 360 horizontal elements. This means that only patterns with 1 degree resolution are supported.

Phase3D

double ** ***WinProp_Pattern::Phase3D***
Array with phase values (rad) for 3D pattern. This array must have the exact size of 180 vertical elements and 360 horizontal elements. This means that only patterns with 1 degree resolution are supported.

Resolution3D

WINPROP_PATTERN_RESOLUTION ***WinProp_Pattern::Resolution3D***
Pattern resolution in degree, only relevant for patterns read from file (Mode is set to PATTERN_MODE_FILE).See also WINPROP_PATTERN_RESOLUTION

AntennaArray

WinProp_Antenna_Array * ***WinProp_Pattern::AntennaArray***
Antenna Array definition for 5G beam steering, only relevant for network planning with WINPROP_PATTERN_MODE = PATTERN_MODE_ANTENNA_ARRAY.

The documentation was generated from the following file:

- `source/Interface/WP_Pattern.h`

2.15.64 WinProp_Polygon

Data Fields

int
NrPoints

COORDPOINT *
Points

int
NrIDs

*int **
IDs

Detailed Description

This structure is used to define a number of database polygons in which the objects inside it will only be considered. It is a part of WinProp_Additional for indoor scenarios and a part of WinProp_ParaMain for urban scenarios.

Field Documentation

NrPoints

int ***WinProp_Polygon::NrPoints***
Number of points in the polygon

Points

COORDPOINT * ***WinProp_Polygon::Points***
The points of the polygon

NrIDs

int ***WinProp_Polygon::NrIDs***
Number of transmitters' Ids

IDs

*int ** ***WinProp_Polygon::IDs***
The Ids of the transmitters which are assigned to the polygon.

- -1 = Assigned to all transmitters.

The documentation was generated from the following file:

- `source/Interface/WP_Polygon.h`

2.15.65 WinProp_PrePro

Data Fields

int
TotalDatabase

COORDPOINT
LowerLeft

COORDPOINT
UpperRight

double
Resolution

int
NrHeights

*double **
Heights

double
TileLength

double
TileLengthSurface

double
TileLengthPlane

double
SegmentLength

int
AdaptiveResolution

int
AdaptiveResolutionFactor

int
SphericalMode

double
SphericalRadius

int
MultipleInteractions

int
OnlyRaysInsideBuilding

int
OnlyPixelsInsideBuilding

int
ExcludeDiffractions

int
MultiThreading

int
PolygonNrCorners

COORDPOINT *
PolygonCorners

int
PreprocessingMode

int
Model

int
TimeVariantMode

double
TimeVariantStart

double
TimeVariantStop

double
TimeVariantInterval

int
TimeVariantNrSteps

*double **
TimeVariantSteps

Detailed Description

This structure is used to preprocess an indoor vector database for the usage with the Intelligent Ray Tracing model. The structure is passed to the function WinProp_PreProcessIndoor.

Remark Important: The structure is packed with 1 byte alignment. It can be initialized with default values by using WinProp_Structure_Init_PrePro.

Field Documentation

TotalDatabase

int ***WinProp_PrePro::TotalDatabase***

- 1 = Preprocess total database
- 0 = Preprocess only a part of the database If only a part of the database is preprocessed, the relevant area has to be with LowerLeft and UpperRight (see below).

LowerLeft

COORDPOINT ***WinProp_PrePro::LowerLeft***
Definition of lower left corner of area (only relevant if TotalDatabase is 0).

UpperRight

COORDPOINT ***WinProp_PrePro::UpperRight***
Definition of upper right corner of area (only relevant if TotalDatabase is 0).

Resolution

double ***WinProp_PrePro::Resolution***
Resolution for preprocessing (is the same as the resolution of prediction later).

NrHeights

int ***WinProp_PrePro::NrHeights***
Number of heights to be preprocessed. This value must be at least 1.

Heights

*double ** ***WinProp_PrePro::Heights***
Array with all height values for which a preprocessing should be computed. There must be at least one height available.

TileLength

double ***WinProp_PrePro::TileLength***
Length of tiles for preprocessing.

TileLengthSurface

double ***WinProp_PrePro::TileLengthSurface***
Length of tiles for preprocessing. Value for surface predictions.

TileLengthPlane

double ***WinProp_PrePro::TileLengthPlane***
Length of tiles for preprocessing. Value for predictions on prediction planes.

SegmentLength

double ***WinProp_PrePro::SegmentLength***
Length of segments for preprocessing.

AdaptiveResolution

int ***WinProp_PrePro::AdaptiveResolution***
Use adaptive resolution?

- 1 = Yes.
- 0 = No.

AdaptiveResolutionFactor

int ***WinProp_PrePro::AdaptiveResolutionFactor***
Factor for adaptive resolution (if used).

SphericalMode

int ***WinProp_PrePro::SphericalMode***
Mode for spherical zone.

- 1 = use fixed spherical zone (radius defined by SphericalRadius).
- 0 = spherical zone not used.

SphericalRadius

double ***WinProp_PrePro::SphericalRadius***
Radius for spherical zone (if used).

MultipleInteractions

int ***WinProp_PrePro::MultipleInteractions***
Consider multiple interactions per ray.

OnlyRaysInsideBuilding

int ***WinProp_PrePro::OnlyRaysInsideBuilding***
Preprocess only rays inside building. There must be a ground plate available in order to distinguish between outdoor and indoor.

OnlyPixelsInsideBuilding

int ***WinProp_PrePro::OnlyPixelsInsideBuilding***
Preprocess only pixels inside building. There must be a ground plate available in order to distinguish between outdoor and indoor.

ExcludeDiffractions

int ***WinProp_PrePro::ExcludeDiffractions***

- 0 = Consider diffractions
- 1 = Diffractions not considered

MultiThreading

int ***WinProp_PrePro::MultiThreading***
Number of threads being used for the preprocessing. Needs to be greater than zero.

PolygonNrCorners

int ***WinProp_PrePro::PolygonNrCorners***
Number of corners in polygon.

PolygonCorners

COORDPOINT * ***WinProp_PrePro::PolygonCorners***
The corners of the polygon.

PreprocessingMode

int ***WinProp_PrePro::PreprocessingMode***
The preprocessing mode.

- 0 = Rectangular area.
- 1 = Area defined by a polygon.
- 2 = Point mode.

Model

int ***WinProp_PrePro::Model***
The propagation model, for which the preprocessing shall be computed, see WINPROP_MODEL_... defines

TimeVariantMode

int ***WinProp_PrePro::TimeVariantMode***
Time-variant mode.

- 0 = Stationary simulation.
- 1 = Simulation with time interval.
- 2 = Simulation with arbitrary time steps.

TimeVariantStart

double ***WinProp_PrePro::TimeVariantStart***
Start time in sec if simulation with time interval

TimeVariantStop

double ***WinProp_PrePro::TimeVariantStop***
End time in sec if simulation with time interval

TimeVariantInterval

double ***WinProp_PrePro::TimeVariantInterval***
Interval in sec between start and end time if simulation with time interval

TimeVariantNrSteps

int ***WinProp_PrePro::TimeVariantNrSteps***
Number of descrete time steps if simulation with arbitrary time steps

TimeVariantSteps

double * ***WinProp_PrePro::TimeVariantSteps***
Descrete time steps in sec if simulation with arbitary time steps

The documentation was generated from the following file:

- `source/Interface/Prepro.h`

2.15.66 WinProp_PreProUrban

Data Fields

PREDMODEL_URBAN
Model

WINPROP_URBAN_IRT_MODE
Mode

int
NrCornersPolygon

int
NrCornersIRTPolygon

COORDPOINT *
CornersPolygon

COORDPOINT *
CornersIRTPolygon

COORDPOINT
lowerLeft

COORDPOINT
upperRight

double
Resolution

int
NrHeights

double *
Heights

int
HeightAbsolute

int
MultipleInteractions

int
IRTPointMode

double
TileVertical

double
TileHorizontal

double
SegmentVertical

double
SegmentHorizontal

int
AdaptiveResolution

WINPROP_URBAN_IRT_ACC_ZONE
SphericMode

double
SphericRadiusConstant

double
SphericRadiusConstantHH

double
SphericRadiusBasic

double
SphericRadiusIncrement

int
ConsiderIndoorPixels

int
ConsiderTopography

int
AbsoluteBuildingHeights

WinProp_Filename
FilenameBuildings

WinProp_Filename
FilenameTopography

int
CNPIndoorMode

float
CNPIndoorTile

float
CNPIndoorSegment

float
CNPUrbanTile

float
CNPResolution

int
CNPHeightMode

int
CNPNrHeights

double *
CNPHeights

int
CNPRedRes

int
CNPRedResFactor

int
CNPSphericZone

float
CNPSphericRadius

int
MultiThreading

Detailed Description

This structure is used to configure preprocessing of an urban vector building database. This structure is passed to the function OutdoorPlugIn_ComputePrePro.

Field Documentation

Model

PREDMODEL_URBAN ***WinProp_PreProUrban::Model***

Selection of propagation model for which the preprocessing will be done.

- PREDMODEL_COST : COST 231 Walfisch-Ikegami model.
- PREDMODEL_UDP : Urban Dominant Path model.
- PREDMODEL_IRT : 3D Intelligent Ray Tracing.

Mode

WINPROP_URBAN_IRT_MODE ***WinProp_PreProUrban::Mode***

Selection of mode for preprocessing. Only relevant if preprocessing is done for the IRT model.

NrCornersPolygon

int ***WinProp_PreProUrban::NrCornersPolygon***

Number of corners for polygon which is used to define the preprocessing area.

NrCornersIRTPolygon

int ***WinProp_PreProUrban::NrCornersIRTPolygon***

Number of corners for polygon which is used to define the polygonal preprocessing sub-area for IRT.

CornersPolygon

COORDPOINT * ***WinProp_PreProUrban::CornersPolygon***

Array with corners of user defined polygon. Number of elements in array is equal to NrCornersPolygon

CornersIRTPolygon

COORDPOINT * ***WinProp_PreProUrban::CornersIRTPolygon***

Array with corners of polygon of preprocessing sub-area. Number of elements in array is equal to NrCornersIRTPolygon

lowerLeft

COORDPOINT ***WinProp_PreProUrban::lowerLeft***

The lower left corner of the preprocessing area, in case no polygons are specified

upperRight

COORDPOINT ***WinProp_PreProUrban::upperRight***

The upper right corner of the preprocessing area, in case no polygons are specified

Resolution

double ***WinProp_PreProUrban::Resolution***
Resolution of result matrix in meters

NrHeights

int ***WinProp_PreProUrban::NrHeights***
Number of individual heights to be preprocessed

Heights

double * ***WinProp_PreProUrban::Heights***
Heights for preprocessing in meters

HeightAbsolute

int ***WinProp_PreProUrban::HeightAbsolute***
Preprocessing absolute or relative: 0 = relative to ground; 1 = absolute (incl. topo)

MultipleInteractions

int ***WinProp_PreProUrban::MultipleInteractions***

- 1 = Consider multiple interactions in one ray. Only relevant if preprocessing is done for the IRT model.

IRTPointMode

int ***WinProp_PreProUrban::IRTPointMode***

- 1 = Preprocess database for IRT in point-mode.
- 0 = Area-wide mode. This includes settings such as polygonal-area preprocessing.

TileVertical

double ***WinProp_PreProUrban::TileVertical***
Vertical size of tiles (meter). Only relevant if preprocessing is done for the IRT model.

TileHorizontal

double ***WinProp_PreProUrban::TileHorizontal***
Horizontal size of tiles (meter). Only relevant if preprocessing is done for the IRT model.

SegmentVertical

double ***WinProp_PreProUrban::SegmentVertical***
Length of vertical segments (meter). Only relevant if preprocessing is done for the IRT model.

SegmentHorizontal

double ***WinProp_PreProUrban::SegmentHorizontal***
Length of horizontal segments (meter). Only relevant if preprocessing is done for the IRT model.

AdaptiveResolution

int ***WinProp_PreProUrban::AdaptiveResolution***
Selection of adaptive resolution (Only relevant if preprocessing is done for the IRT model)

- 0 = no adaptive resolution
- other value indicates the factor for adaptive resolution.

SphericMode

WINPROP_URBAN_IRT_ACC_ZONE ***WinProp_PreProUrban::SphericMode***
Spheric zone can be used for acceleration (Only relevant if preprocessing is done for the IRT model).

SphericRadiusConstant

double ***WinProp_PreProUrban::SphericRadiusConstant***
Radius (meter) of spheric zone for all except relations between horizontal segments. Only relevant if SphericMode is set to PP_MODE_IRT_ZONE_CONSTANT.

SphericRadiusConstantHH

double ***WinProp_PreProUrban::SphericRadiusConstantHH***
Radius (meter) of spheric zone for relations between horizontal segments. Only relevant if SphericMode is set to PP_MODE_IRT_ZONE_CONSTANT or PP_MODE_IRT_ZONE_ADAPTIVE.

SphericRadiusBasic

double ***WinProp_PreProUrban::SphericRadiusBasic***
Basic radius for spheric zone. Only relevant if SphericMode is set to PP_MODE_IRT_ZONE_ADAPTIVE.

SphericRadiusIncrement

double ***WinProp_PreProUrban::SphericRadiusIncrement***
Radius increment (meter per meter) for spheric zone. Only relevant if SphericMode is set to PP_MODE_IRT_ZONE_ADAPTIVE.

ConsiderIndoorPixels

int ***WinProp_PreProUrban::ConsiderIndoorPixels***
Preprocess additionally indoor pixels for IRT indoor coverage. Only relevant in case Model is set to PREDMODEL_IRT.

ConsiderTopography

int ***WinProp_PreProUrban::ConsiderTopography***
Consider topography?

- 1 = topography considered
- 0 = topography not considered

AbsoluteBuildingHeights

int ***WinProp_PreProUrban::AbsoluteBuildingHeights***
Consideration of heights of buildings with respect to topography (Only relevant if topography is considered)

- 1 = absolute heights considered
- 0 = relative heights considered.

FilenameBuildings

WinProp_Filename ***WinProp_PreProUrban::FilenameBuildings***
Filename of building vector database if buildings are not passed via RAM to the API. Needs to be *.odb file, specified without extension.

FilenameTopography

WinProp_Filename ***WinProp_PreProUrban::FilenameTopography***
Filename of topographical database if topography is not passed via RAM to the API.

CNPIndoorMode

int ***WinProp_PreProUrban::CNPIndoorMode***
Combined Network Planning (CNP) preprocessing mode

- 0 = CNP: preprocessing disabled
- 1 = CNP: Indoor coverage empirical (COST or DPM)
- 2 = CNP: Indoor coverage ray-optical (3D IRT)

CNPIndoorTile

float ***WinProp_PreProUrban::CNPIndoorTile***
CNP IRT: Indoor database subdivision tile size (in meters)

CNPIndoorSegment

float ***WinProp_PreProUrban::CNPIndoorSegment***
CNP IRT: Indoor database subdivision segment size (in meters)

CNPUrbanTile

float ***WinProp_PreProUrban::CNPUrbanTile***
CNP IRT: Urban CNP Buildings tile size (in meters)

CNPResolution

float ***WinProp_PreProUrban::CNPResolution***
CNP IRT: Prediction area resolution (in meters)

CNPHeightMode

int ***WinProp_PreProUrban::CNPHeightMode***
CNP IRT: Indoor prediction heights mode

- 0 = Height is identical to urban
- 1 = Absolute individual heights per building

- 2 = Heights relative to the floors

CNPNrHeights

int ***WinProp_PreProUrban::CNPNrHeights***
CNP IRT: Number of prediction heights inside the building of interest

CNPHeights

*double ** ***WinProp_PreProUrban::CNPHeights***
CNP IRT: Individual prediction heights

CNPRedRes

int ***WinProp_PreProUrban::CNPRedRes***
CNP IRT: Adaptive Resolution Management for indoor IRT accelaration

- 1 = enable adaptive resolution
- 0 = disable adaptive resolution

CNPRedResFactor

int ***WinProp_PreProUrban::CNPRedResFactor***
Adaptive resolution factor

CNPSphericZone

int ***WinProp_PreProUrban::CNPSphericZone***
CNP IRT: Spheric Zone mode

- 0 = Spheric Zone OFF
- 1 = Spheric Zone ON (fixed size)
- 2 = Spheric Zone ON (adaptive size)

CNPSphericRadius

float ***WinProp_PreProUrban::CNPSphericRadius***
Spheric Zone fixed size value (in meters)

MultiThreading

int ***WinProp_PreProUrban::MultiThreading***
Number of threads being used for the preprocessing. Needs to be greater than zero.

The documentation was generated from the following file:

- `source/Interface/WP_PreProUrban.h`

2.15.67 WinProp_Propagation_Results

Data Fields

int
FieldStrength

int
PathLoss

int
Delay

int
DelaySpread

int
AngularSpreadMS

int
AngularSpreadBS

int
AngularMeans

int
StatusLOS

int
RayFilePropPaths

int
StrFilePropPaths

int
StrFileCIR

int
StrFileTransMatrix

int
MinNumberOfInteractions

int
ElevationHeights

int
MorphoClassReceiver

int
MorphoAttenuationReceiver

WinProp_Filename
ResultPath

int
AdditionalResultsASCII

Detailed Description

This structure defines the results to be computed and written in binary WinProp format for the WinProp_Predict function. Passing the structure is optional. The power result is always written.

Field Documentation

FieldStrength

int ***WinProp_Propagation_Results::FieldStrength***
FieldStrength as additional result

- 0: disabled
- 1: enabled

PathLoss

int ***WinProp_Propagation_Results::PathLoss***
PathLoss (in dB) as additional result

- 0: disabled
- 1: enabled

Delay

int ***WinProp_Propagation_Results::Delay***
Min. Delay (in ns) as additional result

- 0: disabled
- 1: enabled

DelaySpread

int ***WinProp_Propagation_Results::DelaySpread***
Delay Spread (in ns) as additional result

- 0: disabled
- 1: enabled

AngularSpreadMS

int ***WinProp_Propagation_Results::AngularSpreadMS***
Angular Spread at location of mobile station as additional result

- 0: disabled
- 1: enabled

AngularSpreadBS

int ***WinProp_Propagation_Results::AngularSpreadBS***
Angular Spread at location of base station as additional result

- 0: disabled

- 1: enabled

AngularMeans

int ***WinProp_Propagation_Results::AngularMeans***
Angular means (BS + MS, Azimuth + Elevation) as additional result

- 0: disabled
- 1: enabled

StatusLOS

int ***WinProp_Propagation_Results::StatusLOS***
Status LOS (LOS/OLOS/NLOS) as additional result

- 0: disabled
- 1: enabled

RayFilePropPaths

int ***WinProp_Propagation_Results::RayFilePropPaths***
Propagation paths in .ray file as additional result

- 0: disabled
- 1: enabled

StrFilePropPaths

int ***WinProp_Propagation_Results::StrFilePropPaths***
Propagation paths in .str file as additional result

- 0: disabled
- 1: enabled

StrFileCIR

int ***WinProp_Propagation_Results::StrFileCIR***
Channel impulse response (CIR) in .str file as additional result

- 0: disabled
- 1: enabled

StrFileTransMatrix

int ***WinProp_Propagation_Results::StrFileTransMatrix***
Transmission matrix in .str file as additional result

- 0: disabled
- 1: enabled

MinNumberOfInteractions

int ***WinProp_Propagation_Results::MinNumberOfInteractions***
MinNumberOfInteractions as additional result (only for rural scenarios)

- 0: disabled

- 1: enabled

ElevationHeights

int ***WinProp_Propagation_Results::ElevationHeights***
Elevation relative to Tx including curvature as additional result (only for rural scenarios)

- 0: disabled
- 1: enabled

MorphoClassReceiver

int ***WinProp_Propagation_Results::MorphoClassReceiver***
Morpho class at Rx location (ID) as additional result (only for rural scenarios)

- 0: disabled
- 1: enabled

MorphoAttenuationReceiver

int ***WinProp_Propagation_Results::MorphoAttenuationReceiver***
Morpho attenuation at Rx location (in dB) as additional result (only for rural scenarios)

- 0: disabled
- 1: enabled

ResultPath

WinProp_Filename ***WinProp_Propagation_Results::ResultPath***
Path to which additional results are written

AdditionalResultsASCII

int ***WinProp_Propagation_Results::AdditionalResultsASCII***
Additional results in ASCII format

- 0: disabled
- 1: enabled

The documentation was generated from the following file:

- `source/Interface/WP_Propagation.h`

2.15.68 WinProp_Ray

Data Fields

float
FieldStrength

double
Delay

float
DopplerShift

float
Phase

float
AngleAzimutBTS

float
AngleAzimutMS

float
AngleElevationBTS

float
AngleElevationMS

FIELD_VECTORS
FieldVectors

int
NrInteractions

WinProp_RayInteraction *
Interactions

Detailed Description

This structure contains information about a computed ray. It is returned in the WinProp_ResultPoint structure.

Field Documentation

FieldStrength

float ***WinProp_Ray::FieldStrength***
Computed field strength in uV/m

Delay

double ***WinProp_Ray::Delay***
The delay of the ray in nanoseconds

DopplerShift

float ***WinProp_Ray::DopplerShift***
The Doppler shift in Hz

Phase

float ***WinProp_Ray::Phase***
The phase of the ray in radians

AngleAzimutBTS

float ***WinProp_Ray::AngleAzimutBTS***
The azimuth departure angle at the base station (from East = 0Â° over North = 90Â°), value in degree between [0, 360]

AngleAzimutMS

float ***WinProp_Ray::AngleAzimutMS***
The azimuth incident angle at the mobile station (from East = 0Â° over North = 90Â°), value in degree between [0, 360]

AngleElevationBTS

float ***WinProp_Ray::AngleElevationBTS***
The elevation departure angle at the base station (from Sky = 0Â°, Horizon = 90Â°, Bottom = 180Â°), value in degree between [0, 180]

AngleElevationMS

float ***WinProp_Ray::AngleElevationMS***
The elevation incident angle at the mobile station (from Sky = 0Â°, Horizon = 90Â°, Bottom = 180Â°), value in degree between [0, 180]

FieldVectors

FIELD_VECTORS ***WinProp_Ray::FieldVectors***
The complex field vectors of linear field strength (uV/m)

NrInteractions

int ***WinProp_Ray::NrInteractions***
Number of interactions in ray

Interactions

WinProp_RayInteraction * ***WinProp_Ray::Interactions***
Array (size of NrInteractions) with information about all interaction points

The documentation was generated from the following file:

- `source/Interface/WP_Result.h`

2.15.69 WinProp_RayInteraction

Data Fields

COORDPOINT
Location

WINPROP_RAY_INTERACTION_TYPE
Type

int
Material

int
ObjectID

Detailed Description

This structure contains information about interactions computed for a ray. It is part of the WinProp_Ray structure.

Field Documentation

Location

COORDPOINT ***WinProp_RayInteraction::Location***
Location of interaction (x, y, z)

Type

WINPROP_RAY_INTERACTION_TYPE ***WinProp_RayInteraction::Type***
Type of interaction. Note that not interaction types are supported by all propagation models.

Material

int ***WinProp_RayInteraction::Material***
Index of material on which interaction occurs

ObjectID

int ***WinProp_RayInteraction::ObjectID***
ID of object at which interaction occurs

The documentation was generated from the following file:

- `source/Interface/WP_Result.h`

2.15.70 WinProp_RayMatrix

Data Fields

double *
TimeSteps

WinProp_Dynamic_Antenna *
TransmitterLocations

double
TransmitterTopoHeight

WINPROP_SCENARIO
Scenario

double
Resolution

double
Resolution2

double
LowerLeftX

double
LowerLeftY

double
LowerLeftZ

int
NrHeights

double *
Heights

int
Columns

int
Lines

int
InteractionsEnabled

int
ChannelInformationEnabled

int
DopplerEnabled

int
FieldVectorsEnabled

int
FullPolarizationEvaluated

WinProp_RayMatrixElement ***
Rays

float **
TopoHeight

int
PlaneID

int
UniqueID

WINPROP_PRED_PLANE_TYPE
PlaneType

COORDPOINT
Axis1

COORDPOINT
Axis2

COORDPOINT ***
PixelLocations

Detailed Description

This structure is used to return the computed propagation paths to the calling application.

Field Documentation

TimeSteps

double * ***WinProp_RayMatrix::TimeSteps***
The time step for which this result is valid. This is an array of size equal to NrHeights.

TransmitterLocations

WinProp_Dynamic_Antenna * ***WinProp_RayMatrix::TransmitterLocations***
The transmitter locations and dynamics at the given time step in TimeSteps. This is an array of size equal to NrHeights.

TransmitterTopoHeight

double ***WinProp_RayMatrix::TransmitterTopoHeight***
Topography height at the transmitter location for t=0

Scenario

WINPROP_SCENARIO ***WinProp_RayMatrix::Scenario***
Type of scenario (urban, indoor, rural)

Resolution

double ***WinProp_RayMatrix::Resolution***
Resolution of result in meter (along axis 1)

Resolution2

double ***WinProp_RayMatrix::Resolution2***
Resolution of result in meter (along axis 2)

LowerLeftX

double ***WinProp_RayMatrix::LowerLeftX***
Lower left corner in meter (x-coordinate)

LowerLeftY

double ***WinProp_RayMatrix::LowerLeftY***
Lower left corner in meter (y-coordinate)

LowerLeftZ

double ***WinProp_RayMatrix::LowerLeftZ***
Lower left corner in meter (z-coordinate)

NrHeights

int ***WinProp_RayMatrix::NrHeights***
Number of heights in prediction result

Heights

double * ***WinProp_RayMatrix::Heights***
Array with heights. This is an array of size equal to NrHeights.

Columns

int ***WinProp_RayMatrix::Columns***
Number of columns

Lines

int ***WinProp_RayMatrix::Lines***
Number of lines

InteractionsEnabled

int ***WinProp_RayMatrix::InteractionsEnabled***
Interactions enabled

- 0 = No
- 1 = yes
- 2 = yes with additional info (interaction type, objectID and MaterialID)

ChannelInformationEnabled

int ***WinProp_RayMatrix::ChannelInformationEnabled***
Channel information (phase, azimuth / elevation angles) valid (1) or not (0)

DopplerEnabled

int ***WinProp_RayMatrix::DopplerEnabled***
Doppler shift enabled during computation (1) or disabled (0)

FieldVectorsEnabled

int ***WinProp_RayMatrix::FieldVectorsEnabled***
Field vectors valid (1) or not (0)

FullPolarizationEvaluated

int ***WinProp_RayMatrix::FullPolarizationEvaluated***
Full polarimetric computation (1) or disabled (0)

Rays

WinProp_RayMatrixElement *** ***WinProp_RayMatrix::Rays***
Rays for each pixel, 3D array with size [NrHeights][Columns][Lines]

TopoHeight

float ** ***WinProp_RayMatrix::TopoHeight***
Topography height at each x/y pixel location, 2D array with size [Columns][Lines]

PlaneID

int ***WinProp_RayMatrix::PlaneID***
Identifier for the prediction plane

- CNP: Building index
- Urban Surfaces: Surface number (roof == 0, following the side walls)
- Indoor Surfaces: Number of original wall times 10, plus 1 for surface in front of wall
- Indoor Planes: Number of prediction plane wall

UniqueID

int ***WinProp_RayMatrix::UniqueID***
A unique number, identifying a set of planes.

- In indoor scenarios this is set to a database unique id.
- In case of CNP prediction planes (urban) or surface prediction planes (urban) is set to the building number.

PlaneType

WINPROP_PRED_PLANE_TYPE ***WinProp_RayMatrix::PlaneType***
Type of the plane, see WINPROP_PRED_PLANE_TYPE_PLANE

Axis1

COORDPOINT ***WinProp_RayMatrix::Axis1***
The first axis

Axis2

COORDPOINT ***WinProp_RayMatrix::Axis2***
The second axis

PixelLocations

COORDPOINT *** ***WinProp_RayMatrix::PixelLocations***
Exact location of each pixel, 3D array with size [NrHeights][Columns][Lines]

The documentation was generated from the following file:

- `source/Interface/WP_Result.h`

2.15.71 WinProp_RayMatrixElement

Data Fields

int
NrRays

WinProp_Ray *
Rays

WINPROP_RAY_PIXEL_TYPE
PixelType

PROP_RESULT_PATHTYPES_T
StatusLOS

Detailed Description

This structure contains all computed propagation paths (rays) for a specific matrix pixel.

Field Documentation

NrRays

int ***WinProp_RayMatrixElement::NrRays***
Number of rays computed for the current pixel.

Rays

WinProp_Ray * ***WinProp_RayMatrixElement::Rays***
Array with all rays (see WinProp_Ray) computed for current pixel.

PixelType

WINPROP_RAY_PIXEL_TYPE ***WinProp_RayMatrixElement::PixelType***
Type of the pixel.

StatusLOS

PROP_RESULT_PATHTYPES_T ***WinProp_RayMatrixElement::StatusLOS***
Status of visibility from the transmitter to the current location.

The documentation was generated from the following file:

- `source/Interface/WP_Result.h`

2.15.72 WinProp_RayMatrixList

Data Fields

int
NrRayMatrices

WinProp_RayMatrix *
RayMatrices

Detailed Description

This structure contains a list of ray matrices.

Field Documentation

NrRayMatrices

int ***WinProp_RayMatrixList::NrRayMatrices***
The number of ray matrices

RayMatrices

WinProp_RayMatrix * ***WinProp_RayMatrixList::RayMatrices***
The ray matrices. An array of the length of NrRayMatrices

The documentation was generated from the following file:

- `source/Interface/WP_Result.h`

2.15.73 WinProp_Receiver

Data Fields

int
Enabled

COORDPOINT
Location

WinProp_Pattern *
Pattern

WinProp_Pattern *
PatternPolHoriz

double
AngleYaw

double
AnglePitch

double
AngleRoll

COORDPOINT
Direction

int
HeightAbsolute

int
GroupID

const WinProp_Trajectory *
Trajectory

double
PointSize

int
ReceiverMode

Detailed Description

Structure for receiver configuration if point mode is on.

Orientation of Angles The orientation of the angles and the transformation from the world frame into the body frame of the receiver is as follows:

- Yaw angle (heading): Psi
- Pitch angle (inclination angle): Theta
- Roll angle (bank angle): Phi

In the following index g means global (world) frame and index f means body frame. Rotate the XYZ-system about the z-axis (zg-axis) by Psi. x-axis (xg-axis) is now k1 and y- axis (yg-axis) is now k2. Rotate the XYZ-system about the now rotated y-axis (k2-axis) by Theta. k1-axis is now xf-axis and zg-axis is now k3-axis. Rotate the XYZ-system a third time about the new x-axis (xf-axis) by Phi. k2-axis is now yf- axis and k3-axis is now zf-axis. Thus the sequence of rotation is: Psi > Theta > Phi

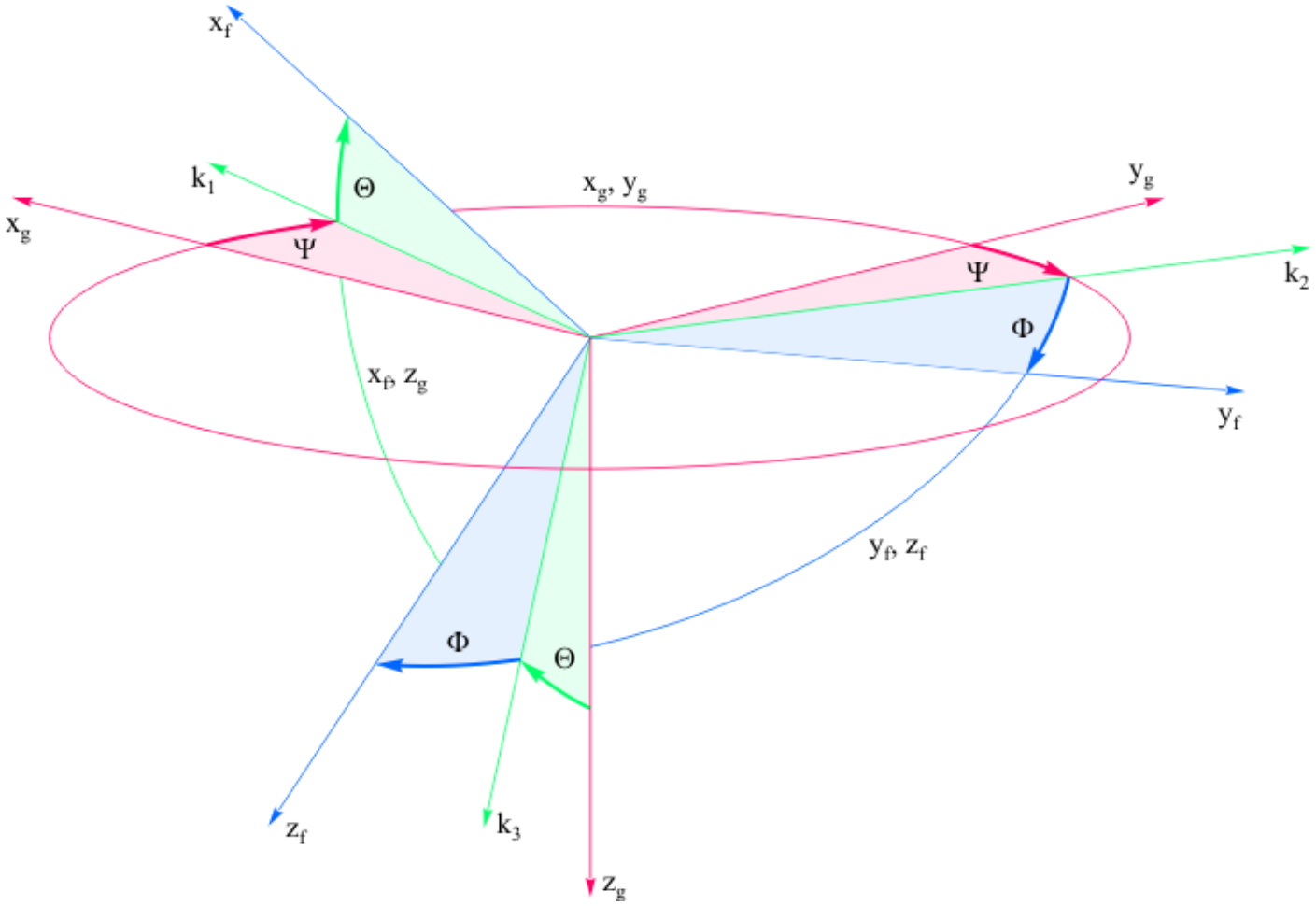

Figure 8: Sequence of rotations: zg - k2 - xf

Field Documentation

Enabled

int ***WinProp_Receiver::Enabled***
Receiver status (enabled/disabled)

- 0 = disabled
- 1 = enabled

Location

COORDPOINT ***WinProp_Receiver::Location***
Location of receiver (x, y, z)

Pattern

WinProp_Pattern * ***WinProp_Receiver::Pattern***
Pattern of receiving antenna (if NULL, no pattern is considered)

PatternPolHoriz

WinProp_Pattern * ***WinProp_Receiver::PatternPolHoriz***
Horizontal pattern of Receiver. Can be optionally defined.

AngleYaw

double ***WinProp_Receiver::AngleYaw***
Yaw angle (degrees)

AnglePitch

double ***WinProp_Receiver::AnglePitch***
Pitch angle (degrees)

AngleRoll

double ***WinProp_Receiver::AngleRoll***
Roll angle (degrees)

Direction

COORDPOINT ***WinProp_Receiver::Direction***
Direction of movement (x,y,z)

HeightAbsolute

int ***WinProp_Receiver::HeightAbsolute***
Height definition of receiving antenna, needs to be identical for all points provided to WinProp_Predict_Points.

- 0 = Height relative to ground
- 1 = Height absolute to sea level
- 2 = Height relative to floors (only in case of indoor scenarios)

GroupID

int ***WinProp_Receiver::GroupID***
GroupID definition in case of moving receiving antenna (only in case of indoor time-variant scenarios).

- 1 = stationary 0 = ID of moving group together which receiving antenna is moving

Trajectory

const WinProp_Trajectory * ***WinProp_Receiver::Trajectory***
Trajectory definition in case of moving receiving antenna along a trajectory. (only in case of indoor time-variant scenarios).

PointSize

double ***WinProp_Receiver::PointSize***
Size of the point, needs to be identical for all points provided to WinProp_Predict_Points. Defines resolution of prediction grid for DPM. For other propagation models it has only visual impact.

ReceiverMode

int ***WinProp_Receiver::ReceiverMode***
Receiving point mode

- 0 = Represents a prediction point
- 1 = Represents a repeater antenna
- 2 = Represents a user
- 3 = Represents a reference point

The documentation was generated from the following file:

- `source/Interface/WP_Receiver.h`

2.15.74 WinProp_Result

Data Fields

WINPROP_RESULT_TYPE
DataType

int
DataTransmissionMode

int
DataCarrierID

double
Resolution

double
LowerLeftX

double
LowerLeftY

int
NrHeights

*double **
Heights

*double **
TimeSteps

int
Columns

int
Lines

*float ****
Matrix

double
Tx_Power

double
Frequency

void *
CoordSystem

WinProp_Dynamic_Antenna *
TransmitterLocations

Detailed Description

This structure saves prediction results and is used for the mapping of all kinds of prediction results.

Field Documentation

DataType

WINPROP_RESULT_TYPE ***WinProp_Result::DataType***
Type of results.

DataTransmissionMode

int ***WinProp_Result::DataTransmissionMode***
Transmission mode or service to which this result belongs (if it is not a global result). This is only relevant for the network planning (see WinProp_Net_Project_Open).

DataCarrierID

int ***WinProp_Result::DataCarrierID***
Carrier ID in case the result individual to a carrier (if it is not a global result). This is only relevant for the network planning (see WinProp_Net_Project_Open).

Resolution

double ***WinProp_Result::Resolution***
Resolution of result in meter

LowerLeftX

double ***WinProp_Result::LowerLeftX***
Lower left corner in meter (x-coordinate)

LowerLeftY

double ***WinProp_Result::LowerLeftY***
Lower left corner in meter (y-coordinate)

NrHeights

int ***WinProp_Result::NrHeights***
Number of heights or time steps in prediction result

Heights

double * ***WinProp_Result::Heights***
Array with heights. This is an array of size equal to NrHeights.

TimeSteps

double * ***WinProp_Result::TimeSteps***
Array with time steps. This is an array of size equal to NrHeights.

Columns

int ***WinProp_Result::Columns***
Number of columns

Lines

int ***WinProp_Result::Lines***
Number of lines

Matrix

float *** ***WinProp_Result::Matrix***
Result values. A value can be accessed with indices [NrHeights][Columns][Lines]

Tx_Power

double ***WinProp_Result::Tx_Power***
Power (dBm) of transmitter assigned to this result

Frequency

double ***WinProp_Result::Frequency***
Frequency (MHz) of transmitter assigned to this result

CoordSystem

void * ***WinProp_Result::CoordSystem***
Information about coordinate system (optional). Can be set to NULL.

TransmitterLocations

WinProp_Dynamic_Antenna * ***WinProp_Result::TransmitterLocations***
The transmitter locations and dynamics for the various time steps. This is an array of size equal to NrHeights.

The documentation was generated from the following file:

- `source/Interface/WP_Result.h`

2.15.75 WinProp_Result_2

Data Fields

int
NbrElements

*float **
LinearArray

*float **
CoordX

*float **
CoordY

Detailed Description

Structure for alternative result array used in WinProp_Convert_Result

Field Documentation

NbrElements

int ***WinProp_Result_2::NbrElements***
Number of elements inferred from WinProp_Result

LinearArray

*float ** ***WinProp_Result_2::LinearArray***
Values in array obtained from NbrElements

CoordX

*float ** ***WinProp_Result_2::CoordX***
x coordinate of prediction area

CoordY

*float ** ***WinProp_Result_2::CoordY***
y coordinate of prediction area

The documentation was generated from the following file:

- `source/Interface/WP_Result.h`

2.15.76 WinProp_ResultMonteCarlo

Data Fields

unsigned int
nbrApplications

unsigned int
nbrCells

unsigned int
nbrSnapshots

WinProp_TransmitterLoad **
trxLoad

WinProp_MonteCarloStatistics ***
statisticCell

WinProp_MonteCarloStatistics **
statisticTotal

Detailed Description

Statistical results of the Monte-Carlo simulation in the network planning (see WinProp_Net_Project_Open)

Field Documentation

nbrApplications

unsigned int ***WinProp_ResultMonteCarlo::nbrApplications***
Number of applications

nbrCells

unsigned int ***WinProp_ResultMonteCarlo::nbrCells***
Number of cells

nbrSnapshots

unsigned int ***WinProp_ResultMonteCarlo::nbrSnapshots***
Number of Monte-Carlo snapshots

trxLoad

WinProp_TransmitterLoad ** ***WinProp_ResultMonteCarlo::trxLoad***
Load parameters for the transmitter (cell load, noise rise, throughput for cell) [nbrSnapshots][nbrCells]

statisticCell

WinProp_MonteCarloStatistics *** ***WinProp_ResultMonteCarlo::statisticCell***
The statistics of the Monte-Carlo simulation per cell and application [nbrSnapshots][nbrCells][nbrApplications]

statisticTotal

WinProp_MonteCarloStatistics ** ***WinProp_ResultMonteCarlo::statisticTotal***
The statistics of the Monte-Carlo simulation per application [nbrSnapshots][nbrApplications]

The documentation was generated from the following file:

- `source/Interface/WP_Result.h`

2.15.77 WinProp_ResultPlane

Data Fields

int
TimeStepIndex

int
PlaneID

int
UniqueID

int
Columns

int
Lines

WINPROP_PRED_PLANE_TYPE
PlaneType

PLANEPOINT **
Matrix

COORDPOINT
Origin

COORDPOINT
NormalVector

double
HeightOffset

COORDPOINT
Axis1

COORDPOINT
Axis2

double
Resolution1

double
Resolution2

Detailed Description

Structure with results computed in plane. This structure is used for the mapping of all kinds of prediction results on a prediction plane. Prediction planes can be computed with the function WinProp_Predict_Planes.

Field Documentation

TimeStepIndex

int ***WinProp_ResultPlane::TimeStepIndex***
Zero-based index of the timestep

PlaneID

int ***WinProp_ResultPlane::PlaneID***
Identifier for the prediction plane

- CNP: Building index
- Urban Surfaces: Surface number (roof == 0, following the side walls)
- Indoor Surfaces: Number of original wall times 10, plus 1 for surface in front of wall
- Indoor Planes: Number of prediction plane wall

UniqueID

int ***WinProp_ResultPlane::UniqueID***
A unique number, identifying a set of planes.

- In indoor scenarios this is set to a database unique id.
- In case of CNP prediction planes (urban) or surface prediction planes (urban) is set to the building number.

Columns

int ***WinProp_ResultPlane::Columns***
Columns of result matrix

Lines

int ***WinProp_ResultPlane::Lines***
Lines of result matrix

PlaneType

WINPROP_PRED_PLANE_TYPE ***WinProp_ResultPlane::PlaneType***
Type of the plane, see WINPROP_PRED_PLANE_TYPE_PLANE

Matrix

PLANEPOINT ** ***WinProp_ResultPlane::Matrix***
Array (x and y direction) with result information. Result information is available in PLANEPOINT for each pixel.

Origin

COORDPOINT ***WinProp_ResultPlane::Origin***
Origin/reference point of the plane (lower left corner).

NormalVector

COORDPOINT ***WinProp_ResultPlane::NormalVector***
Normal vector of plane/polygon

HeightOffset

double ***WinProp_ResultPlane::HeightOffset***
Offset (meter) in case of topography

Axis1

COORDPOINT ***WinProp_ResultPlane::Axis1***
The first axis

Axis2

COORDPOINT ***WinProp_ResultPlane::Axis2***
The second axis

Resolution1

double ***WinProp_ResultPlane::Resolution1***
Resolution along axis 1

Resolution2

double ***WinProp_ResultPlane::Resolution2***
Resolution along axis 2

The documentation was generated from the following file:

- `source/Interface/WP_Result.h`

2.15.78 WinProp_ResultPlaneList

Data Fields

WINPROP_RESULT_TYPE
DataType

int
DataTransmissionMode

int
DataCarrierID

*void **
CoordSystem

int
NrTimeSteps

double *
TimeSteps

int
NrResultPlanes

WinProp_ResultPlane *
ResultPlanes

double
Tx_Power

double
Frequency

Detailed Description

This structure contains a list of result planes.

Field Documentation

DataType

WINPROP_RESULT_TYPE ***WinProp_ResultPlaneList::DataType***
Type of result in the matrix. See WinProp_Result::DataType

DataTransmissionMode

int ***WinProp_ResultPlaneList::DataTransmissionMode***
Transmission mode or service to which this result belongs (if it is not a global result). This is only relevant for the network planning (see WinProp_Net_Project_Open).

DataCarrierID

int ***WinProp_ResultPlaneList::DataCarrierID***
Carrier ID in case the result individual to a carrier (if it is not a global result). This is only relevant for the network planning (see WinProp_Net_Project_Open).

CoordSystem

void * ***WinProp_ResultPlaneList::CoordSystem***
Information about coordinate system

NrTimeSteps

int ***WinProp_ResultPlaneList::NrTimeSteps***
The number of timesteps

TimeSteps

double * ***WinProp_ResultPlaneList::TimeSteps***
The time step for which this result is valid [NrTimeSteps]

NrResultPlanes

int ***WinProp_ResultPlaneList::NrResultPlanes***
The number of result planes

ResultPlanes

WinProp_ResultPlane * ***WinProp_ResultPlaneList::ResultPlanes***
The results of the individual planes

Tx_Power

double ***WinProp_ResultPlaneList::Tx_Power***
Power (dBm) of transmitter assigned to this result (Relevant for network planning).

Frequency

double ***WinProp_ResultPlaneList::Frequency***
Frequency (MHz) of transmitter assigned to this result (Relevant for network planning).

The documentation was generated from the following file:

- `source/Interface/WP_Result.h`

2.15.79 WinProp_ResultPoint

Data Fields

int
TimeStepIndex

int
receiverIndex

int
PointID

double
ResultValue

int
Computed

COORDPOINT
Location

COORDPOINT
MovingDir

float
Velocity

float
Pitch

float
Roll

float
Yaw

double
TopoHeight

WinProp_RayMatrixElement
Rays

Detailed Description

This structure contains the computed result and propagation paths (rays) for a specific point.

Field Documentation

TimeStepIndex

int ***WinProp_ResultPoint::TimeStepIndex***
Zero-based index of the timestep

receiverIndex

int ***WinProp_ResultPoint::receiverIndex***
Zero-based index of the receiver

PointID

int ***WinProp_ResultPoint::PointID***
Identifier for the point

ResultValue

double ***WinProp_ResultPoint::ResultValue***
Computed field strength (dBuV/m) or path loss (dB) or power (dBm). This depends on WinProp_Result::DataType.

Computed

int ***WinProp_ResultPoint::Computed***
Indicates whether a result has been computer (1) or not (0)

Location

COORDPOINT ***WinProp_ResultPoint::Location***
Location of receiver point (x, y, z)

MovingDir

COORDPOINT ***WinProp_ResultPoint::MovingDir***
Direction of movement

Velocity

float ***WinProp_ResultPoint::Velocity***
The velocity at this point

Pitch

float ***WinProp_ResultPoint::Pitch***
The pitch angle at this point

Roll

float ***WinProp_ResultPoint::Roll***
The roll angle at this point

Yaw

float ***WinProp_ResultPoint::Yaw***
The yaw angle at this point

TopoHeight

double ***WinProp_ResultPoint::TopoHeight***
Topography height at the point location

Rays

WinProp_RayMatrixElement ***WinProp_ResultPoint::Rays***
all computed propagation paths (rays)

The documentation was generated from the following file:

- `source/Interface/WP_Result.h`

2.15.80 WinProp_ResultPointsList

Data Fields

WINPROP_RESULT_TYPE
DataType

int
DataTransmissionMode

int
DataCarrierID

int
NrTimeSteps

*double **
TimeSteps

WinProp_Dynamic_Antenna *
TransmitterLocations

double
TransmitterTopoHeight

double
PointResolution

WINPROP_SCENARIO
Scenario

int
NrResultPoints

WinProp_ResultPoint *
ResultPoints

int
InteractionsEnabled

int
ChannelInformationEnabled

int
DopplerEnabled

int
FieldVectorsEnabled

int
FullPolarizationEvaluated

int
UniqueDBID

Detailed Description

This structure is used for point-to-point predictions. Point-to-point predictions can be computed with the function WinProp_Predict_Points.

Field Documentation

DataType

WINPROP_RESULT_TYPE ***WinProp_ResultPointsList::DataType***
Type of result in the matrix. See WinProp_Result::DataType

DataTransmissionMode

int ***WinProp_ResultPointsList::DataTransmissionMode***
Transmission mode or service to which this result belongs (if it is not a global result). This is only relevant for the network planning (see WinProp_Net_Project_Open).

DataCarrierID

int ***WinProp_ResultPointsList::DataCarrierID***
Carrier ID in case the result individual to a carrier (if it is not a global result). This is only relevant for the network planning (see WinProp_Net_Project_Open).

NrTimeSteps

int ***WinProp_ResultPointsList::NrTimeSteps***
The number of timesteps

TimeSteps

double * ***WinProp_ResultPointsList::TimeSteps***
The time step for which this result is valid [NrTimeSteps]

TransmitterLocations

WinProp_Dynamic_Antenna * ***WinProp_ResultPointsList::TransmitterLocations***
The transmitter locations and dynamics for the various timesteps [NrTimeSteps]

TransmitterTopoHeight

double ***WinProp_ResultPointsList::TransmitterTopoHeight***
Topography height at the transmitter location for t=0

PointResolution

double ***WinProp_ResultPointsList::PointResolution***
The size in x/y-dimension of a single point

Scenario

WINPROP_SCENARIO ***WinProp_ResultPointsList::Scenario***
Type of scenario (urban, indoor, rural)

NrResultPoints

int ***WinProp_ResultPointsList::NrResultPoints***
The number of result points

ResultPoints

WinProp_ResultPoint * ***WinProp_ResultPointsList::ResultPoints***
The results of the individual points

InteractionsEnabled

int ***WinProp_ResultPointsList::InteractionsEnabled***
Interactions enabled

- 0 = No
- 1 = yes
- 2 = yes with additional info (interaction type, objectID and MaterialID)

ChannelInformationEnabled

int ***WinProp_ResultPointsList::ChannelInformationEnabled***
Channel information (phase, azimuth / elevation angles) valid (1) or not (0)

DopplerEnabled

int ***WinProp_ResultPointsList::DopplerEnabled***
Doppler shift enabled during computation (1) or disabled (0)

FieldVectorsEnabled

int ***WinProp_ResultPointsList::FieldVectorsEnabled***
Field vectors valid (1) or not (0)

FullPolarizationEvaluated

int ***WinProp_ResultPointsList::FullPolarizationEvaluated***
Full polarimetric computation (1) or disabled (0)

UniqueDBID

int ***WinProp_ResultPointsList::UniqueDBID***
A unique number, identifying the for the computation used database.

The documentation was generated from the following file:

- `source/Interface/WP_Result.h`

2.15.81 WinProp_ResultTrajectory

Data Fields

int
NrSampledPoints

WinProp_Trajectory_Sample *
SampledPoints

WinProp_ResultPointsList
ResultPoints

Detailed Description

This structure contains the results of a trajectory computation.

Field Documentation

NrSampledPoints

int ***WinProp_ResultTrajectory::NrSampledPoints***
The number of sampled points

SampledPoints

WinProp_Trajectory_Sample * ***WinProp_ResultTrajectory::SampledPoints***
The points of the sampled trajectory. An array of the length of NrSampledPoints

ResultPoints

WinProp_ResultPointsList ***WinProp_ResultTrajectory::ResultPoints***
The computed results. A result point may cover multiple samples.

The documentation was generated from the following file:

- `source/Interface/WP_Result.h`

2.15.82 WinProp_ResultTrajectoryList

Data Fields

int
NrTrajectories

WinProp_ResultTrajectory *
trajectories

Detailed Description

This structure contains the results of a trajectory computation.

Field Documentation

NrTrajectories

int ***WinProp_ResultTrajectoryList::NrTrajectories***
The number of sampled points

trajectories

WinProp_ResultTrajectory * ***WinProp_ResultTrajectoryList::trajectories***
The points of the sampled trajectory. An array of the length of NrSampledPoints

The documentation was generated from the following file:

- `source/Interface/WP_Result.h`

2.15.83 WinProp_SamplingPoint

Data Fields

double
Value

int
Color

Detailed Description

Field Documentation

Value

double ***WinProp_SamplingPoint::Value***

Color

int ***WinProp_SamplingPoint::Color***

The documentation was generated from the following file:

- `source/Interface/Engine.h`

2.15.84 WinProp_Scenario

Data Fields

WINPROP_SCENARIO
Scenario

WinProp_Filename
VectorDatabase

WinProp_Filename
PixelBldDatabase

WinProp_Filename
TopographyDatabase

WinProp_Filename
ClutterDatabase

WinProp_Filename
ClutterTable

int
VectorDatabaseSorting

Detailed Description

This structure is used to define a scenario when calling WinProp_Open.

Field Documentation

Scenario

WINPROP_SCENARIO ***WinProp_Scenario::Scenario***
Type of scenario (urban, indoor, rural)

VectorDatabase

WinProp_Filename ***WinProp_Scenario::VectorDatabase***
Name of the vector database for indoor and urban scenarios

PixelBldDatabase

WinProp_Filename ***WinProp_Scenario::PixelBldDatabase***
Filename of buildings vector database in supported raster/pixel format.

TopographyDatabase

WinProp_Filename ***WinProp_Scenario::TopographyDatabase***
Name of the topography database

ClutterDatabase

WinProp_Filename ***WinProp_Scenario::ClutterDatabase***
Name of clutter database (rural only)

ClutterTable

WinProp_Filename ***WinProp_Scenario::ClutterTable***
Name of clutter table (rural only)

VectorDatabaseSorting

int ***WinProp_Scenario::VectorDatabaseSorting***
Sort walls by size in vector database

- 0 : do NOT sort walls
- 1 : do sort walls

The documentation was generated from the following file:

- `source/Interface/WP_Propagation.h`

2.15.85 WinProp_Surveydata

Data Fields

int
NbrValues

*double **
CoordianteX

*double **
CoordianteY

*double **
CoordianteZ

*double **
Power

Detailed Description

This structure is used to pass measurement data to the wave propagation API. This is done by using WinProp_Coverage_Analysis.

Field Documentation

NbrValues

int ***WinProp_Surveydata::NbrValues***
Number of measurement points

CoordianteX

*double ** ***WinProp_Surveydata::CoordianteX***
Array with x-coordinates of measurement points

CoordianteY

*double ** ***WinProp_Surveydata::CoordianteY***
Array with y-coordinates of measurement points

CoordianteZ

*double ** ***WinProp_Surveydata::CoordianteZ***
Array with z-coordinates of measurement points

Power

*double ** ***WinProp_Surveydata::Power***
Array with power values (dBm) of measurement points

The documentation was generated from the following file:

- `source/Interface/WP_Measurements.h`

2.15.86 WinProp_Trajectory

Data Fields

int
Id

int
Enabled

WinProp_NameString
Name

int
NrPoints

WinProp_Trajectory_Point *
Points

int
samplingMode

double
samplingResolution

int
addPointsToSamples

int
HeightAbsolute

double
PointSize

Detailed Description

Structure for receiver configuration if trajectory mode is on.

Field Documentation

Id

int ***WinProp_Trajectory::Id***
The ID of the trajectory

Enabled

int ***WinProp_Trajectory::Enabled***
The status of the trajectory (enabled/disabled)

- 0 = disabled
- 1 = enabled

Name

WinProp_NameString ***WinProp_Trajectory::Name***
The name of the trajectory

NrPoints

int ***WinProp_Trajectory::NrPoints***
The number of points

Points

WinProp_Trajectory_Point * ***WinProp_Trajectory::Points***
The points defining the trajectory. The first element will be the start of the trajectory, while the last element defines the end. Array of length NrPoints

samplingMode

int ***WinProp_Trajectory::samplingMode***
The sampling mode.

- 0 = Sample the distance between evaluation points.
- 1 = Sample the time between evaluation points.

samplingResolution

double ***WinProp_Trajectory::samplingResolution***
The sampling resolution. Based on samplingMode either in seconds or in meter.

addPointsToSamples

int ***WinProp_Trajectory::addPointsToSamples***
Defines if the in Points defined locations are part of the trajectory samples.

- 1 = the in Points defined locations will be part of the trajectory samples and effectively only between those points the sampling will be done
- 0 = the in Points defined locations will not necessarily be part of the samples

HeightAbsolute

int ***WinProp_Trajectory::HeightAbsolute***
Height definition of receiving antenna.

- 0 = Height relative to ground
- 1 = Height absolute to sea level
- 2 = Height relative to floors (only in case of indoor scenarios)

PointSize

double ***WinProp_Trajectory::PointSize***
Size of the predicted points, needs to be identical for all trajectories provided to WinProp_Predict_Trajectories.

The documentation was generated from the following file:

- `source/Interface/WP_Trajectory.h`

2.15.87 WinProp_Trajectory_Point

Data Fields

COORDPOINT
location

float
velocity

float
pitch

float
roll

float
yaw

Detailed Description

This structure contains the properties of a trajectory point.

Field Documentation

location

COORDPOINT ***WinProp_Trajectory_Point::location***
The location of this point

velocity

float ***WinProp_Trajectory_Point::velocity***
The velocity at this point

pitch

float ***WinProp_Trajectory_Point::pitch***
The pitch angle at this point

roll

float ***WinProp_Trajectory_Point::roll***
The roll angle at this point

yaw

float ***WinProp_Trajectory_Point::yaw***
The yaw angle at this point

The documentation was generated from the following file:

- `source/Interface/WP_Trajectory.h`

2.15.88 WinProp_Trajectory_Sample

Data Fields

WinProp_Trajectory_Point
properties

COORDPOINT
movingDir

float
time

float
distanceToStart

int
ResultPointIndex

Detailed Description

This structure contains information about a specific samples point of a trajectory.

Field Documentation

properties

WinProp_Trajectory_Point ***WinProp_Trajectory_Sample::properties***
The sample properties, e.g. location, orientation and velocity

movingDir

COORDPOINT ***WinProp_Trajectory_Sample::movingDir***
Direction of movement

time

float ***WinProp_Trajectory_Sample::time***
The time elapsed since the start of the trajectory

distanceToStart

float ***WinProp_Trajectory_Sample::distanceToStart***
Distance of this point to the first point of the trajectory

ResultPointIndex

int ***WinProp_Trajectory_Sample::ResultPointIndex***
Zero-based index of the corresponding result point

The documentation was generated from the following file:

- `source/Interface/WP_Result.h`

2.15.89 WinProp_TransmitterLoad

Data Fields

float
LoadPowerDL

float
LoadResourcesDL

float
LoadResourcesUL

float
NoiseRiseUL

float
ThroughputDL

float
ThroughputUL

Detailed Description

Parameters describing the transmitter load after the network planning.

Field Documentation

LoadPowerDL

float ***WinProp_TransmitterLoad::LoadPowerDL***
Percentage of downlink load (TX power)

LoadResourcesDL

float ***WinProp_TransmitterLoad::LoadResourcesDL***
The load resources downlink

LoadResourcesUL

float ***WinProp_TransmitterLoad::LoadResourcesUL***
The load resources uplink

NoiseRiseUL

float ***WinProp_TransmitterLoad::NoiseRiseUL***
Noise rise [dB] in uplink for static analysis

ThroughputDL

float ***WinProp_TransmitterLoad::ThroughputDL***
The downlink throughput

ThroughputUL

float ***WinProp_TransmitterLoad::ThroughputUL***
The uplink throughput

The documentation was generated from the following file:

- `source/Interface/WP_Result.h`

Index

G

I

M

N

O

P

R

S

T

U

V

W