**Energy Model .cpp**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <math.h>

#include <limits.h>

#include <phy.h>

#include "api.h"

#define ENERGY\_DEBUG 1

// FUNCTION: Phy\_ReportStatusToEnergyModel

// LAYER : PHYSICAL

// PURPOSE: This function should be called whenever a state transition occurs

// in any place in PHY layer. As input parameters, the function reads the current

// state and the new state of PHY layer and based on the new sates calculates the cost

// of the load that should be taken off the battery.

// The function then interacts with battery model and updates the charge of battery.

// PARAMETERS:

// +node: Node\*: The node received message

// +phyIndex: index of the interface running this PHY layer

// +prevStatus:the state from which PHY is exiting

// +newStatus: int:the state to which PHY is entering

// RETURN : None

void

Phy\_ReportStatusToEnergyModel(

Node\* node,

const int phyIndex,

PhyStatusType prevStatus,

PhyStatusType newStatus )

{

double duration = 0;

double actDuration = 0;

double load = 0;

PhyData\* thisPhy = node->phyData[phyIndex];

clocktype now = node->getNodeTime();

if (thisPhy->eType == NO\_ENERGY\_MODEL )

{

return;

}

duration = (double)( now - thisPhy->curLoad->lastUpdate ) / (double)SECOND;

load = thisPhy->curLoad->load;

actDuration = (double)( now - thisPhy->curLoad->startTime ) / (double)SECOND;

float sleep\_load = thisPhy->powerConsmpTable->sleep\_current\_load;

// Get number of configured antenna elements

int numConfigAntennas = PHY\_GetNumConfigAntennas(node, phyIndex);

// Get number of active antenna elements

int numActiveAntennas = PHY\_GetNumActiveAntennas(node, phyIndex);

// (numConfigAntennas - numActiveAntennas) reperesent the number of

// antennas which are currently inactive. Inactive antennas are assumed

// to be in sleep mode and consume power as per sleep load configured.

if (!node->battery->dead)

{

// Decrement the battery charge for currently active antennas based

// on current mode.

BatteryDecCharge(node, duration, load \* numActiveAntennas);

// Decrement the battery charge for inactive antennas

BatteryDecCharge(node, duration, sleep\_load

\* (numConfigAntennas - numActiveAntennas));

}

else

{

now = node->battery->deadTime ;

actDuration

= (double)(now - thisPhy->curLoad->startTime) / (double)SECOND;

if (actDuration < 0 )

{

return;

}

}

if (thisPhy->eType == GENERIC\_ENERGY\_MODEL)

{

Generic\_UpdateCurrentLoad(

node,

phyIndex);

}

switch (prevStatus)

{

case PHY\_SUCCESS:

case PHY\_IDLE:

{

thisPhy->curLoad->powStats.totalIdlePower

+= (load \* actDuration \* numActiveAntennas)

+ (sleep\_load \* actDuration

\* (numConfigAntennas - numActiveAntennas));

thisPhy->curLoad->powStats.totalIdleDuration +=

(clocktype )( now - thisPhy->curLoad->startTime );

if (ENERGY\_DEBUG )

{

printf("Node %d:Total Idle charge consumed: %e \n",

node->nodeId,

(thisPhy->curLoad->powStats.totalIdlePower/3600.0));

printf("\n our modification\n");

}

break;

}

case PHY\_BUSY\_TX:

case PHY\_TRANSMITTING:

{

thisPhy->curLoad->powStats.totalTxPower

+= (load \* actDuration \* numActiveAntennas)

+ (sleep\_load \* actDuration

\* (numConfigAntennas - numActiveAntennas));

thisPhy->curLoad->powStats.totalTxDuration +=

(clocktype )( now - thisPhy->curLoad->startTime );

if (ENERGY\_DEBUG )

{

printf("Node %d:Total Transmit charge consumed: %e \n",

node->nodeId,

(thisPhy->curLoad->powStats.totalTxPower/3600.0));

}

break;

}

case PHY\_BUSY\_RX:

case PHY\_SENSING:

case PHY\_RECEIVING:

{

thisPhy->curLoad->powStats.totalRxPower

+= (load \* actDuration \* numActiveAntennas)

+ (sleep\_load \* actDuration

\* (numConfigAntennas - numActiveAntennas));

thisPhy->curLoad->powStats.totalRxDuration +=

(clocktype)( now - thisPhy->curLoad->startTime );

if (ENERGY\_DEBUG )

{

printf("Node %d:Total Receive charge consumed:%e\n",

node->nodeId,

(thisPhy->curLoad->powStats.totalRxPower/3600.0));

}

break;

}

case PHY\_TRX\_OFF:

{

thisPhy->curLoad->powStats.totalSleepPower

+= (load \* actDuration \* numActiveAntennas )

+ (sleep\_load \* actDuration

\* (numConfigAntennas - numActiveAntennas));

thisPhy->curLoad->powStats.totalSleepDuration +=

(clocktype )( now - thisPhy->curLoad->startTime );

if (ENERGY\_DEBUG )

{

printf("Node %d: Total Sleep duration:%e\n",

node->nodeId,

(double)

(thisPhy->curLoad->powStats.totalSleepDuration/SECOND));

}

break;

}

} //switch(prevStatus)

switch (newStatus)

{

case PHY\_SUCCESS:

case PHY\_IDLE:

{

thisPhy->curLoad->load

= thisPhy->powerConsmpTable->idle\_current\_load;

thisPhy->curLoad->startTime = node->getNodeTime();

thisPhy->curLoad->lastUpdate = node->getNodeTime();

break;

}

case PHY\_BUSY\_TX:

case PHY\_TRANSMITTING:

{

thisPhy->curLoad->load

= thisPhy->powerConsmpTable->trx\_current\_load;

thisPhy->curLoad->startTime = node->getNodeTime();

thisPhy->curLoad->lastUpdate = node->getNodeTime();

break;

}

case PHY\_BUSY\_RX:

case PHY\_SENSING:

case PHY\_RECEIVING:

{

thisPhy->curLoad->load

= thisPhy->powerConsmpTable->rcv\_current\_load;

thisPhy->curLoad->startTime = node->getNodeTime();

thisPhy->curLoad->lastUpdate = node->getNodeTime();

break;

}

case PHY\_TRX\_OFF:

{

thisPhy->curLoad->load

= thisPhy->powerConsmpTable->sleep\_current\_load;

thisPhy->curLoad->startTime = node->getNodeTime();

thisPhy->curLoad->lastUpdate = node->getNodeTime();

break;

}

} //switch(newStatus)

if (node->guiOption)

{

GUI\_SendRealData(node->nodeId,

thisPhy->curLoad->RuntimeId,

thisPhy->curLoad->load,

node->getNodeTime());

}

}

// FUNCTION: ENERGY\_Init

// LAYER : PHYSICAL

// PURPOSE: This function declares energy model variables and initializes them.

// Moreover, the function read energy model specifications and configures

// the parameters which are configurable.

// PARAMETERS:

// +node: Node\*: The node received message

// +phyIndex: index of the interface running this PHY layer

// +nodeInput:

// RETURN : None

void

ENERGY\_Init(

Node \*node,

const int phyIndex,

const NodeInput \*nodeInput)

{

PhyData\* thisPhy;

int i;

BOOL found = FALSE;

char str[MAX\_STRING\_LENGTH];

double txPower\_dBm,txPower\_mW;

double load;

thisPhy = node->phyData[phyIndex];

IO\_ReadString(

node->nodeId,

thisPhy->networkAddress,

nodeInput,

"ENERGY-MODEL-SPECIFICATION",

&found,

str);

if (!found || !strcmp(str, "NONE"))

{

thisPhy->eType = NO\_ENERGY\_MODEL;

return;

}

if (ENERGY\_DEBUG){

printf("Node %d:Initiliazing energy model \n",

node->nodeId);

}

thisPhy->curLoad = (LoadProfile\*)

MEM\_malloc(sizeof(LoadProfile));

thisPhy->curLoad->startTime = (clocktype) 0;

thisPhy->curLoad->lastUpdate = (clocktype) 0;

thisPhy->curLoad->load = 0.0;

thisPhy->curLoad->powStats.totalIdlePower = 0.0;

thisPhy->curLoad->powStats.totalSleepPower = 0.0;

thisPhy->curLoad->powStats.totalTxPower = 0.0;

thisPhy->curLoad->powStats.totalRxPower = 0.0;

thisPhy->curLoad->powStats.totalSleepDuration = (clocktype) 0;

thisPhy->curLoad->powStats.totalIdleDuration = (clocktype) 0;

thisPhy->curLoad->powStats.totalRxDuration = (clocktype) 0;

thisPhy->curLoad->powStats.totalTxDuration = (clocktype) 0;

PowerCosts\* loadTable = (PowerCosts\*)

MEM\_malloc(sizeof(PowerCosts));

thisPhy->eType = TECHNOLOGY\_DEFINED\_ENERGY\_MODEL;

loadTable->sleep\_current\_load = 0.0;

loadTable->idle\_current\_load = 5.0;

loadTable->rcv\_current\_load = 10.0;

loadTable->trx\_current\_table = (float\*)

MEM\_malloc((NUM\_TRX\_POWER\_STATES)\*sizeof(float));

for (i = 0; i < NUM\_TRX\_POWER\_STATES; i++){

loadTable->trx\_current\_table[i] = 12.0;

}

int numConfigAntennas = PHY\_GetNumConfigAntennas(node, phyIndex);

if (!strcmp(str, "MICA-MOTES")){

PHY\_GetTransmitPower(

node,

phyIndex,

&txPower\_mW);

txPower\_mW = txPower\_mW/numConfigAntennas;

txPower\_dBm = (double) 10.0 \* (log(txPower\_mW) / log(10.0));

switch (RoundToInt(txPower\_dBm))

{

case 10:

{

loadTable->trx\_current\_load = 26.7f;

break;

}

case 5:

{

loadTable->trx\_current\_load = 14.8f;

break;

}

case 0:

{

loadTable->trx\_current\_load = 10.4f;

break;

}

case -5:

{

loadTable->trx\_current\_load = 8.9f;

break;

}

case -20:

{

loadTable->trx\_current\_load = 5.3f;

break;

}

default:

{

loadTable->trx\_current\_load =

float ((txPower\_mW -1.0)\* 1.14 + 10.4);

}

}

loadTable->rcv\_current\_load = 9.6f;

loadTable->sleep\_current\_load = 0.03f;

loadTable->idle\_current\_load = 5.0;

loadTable->voltage = 3.0;

}else if (!found || !strcmp(str, "MICAZ") ){

PHY\_GetTransmitPower(

node,

phyIndex,

&txPower\_mW);

txPower\_mW = txPower\_mW/numConfigAntennas;

txPower\_dBm = (double) 10.0 \* ( log(txPower\_mW) / log(10.0));

switch (RoundToInt(txPower\_dBm))

{

case 0:

{

loadTable->trx\_current\_load = 16.0;

break;

}

case -1:

{

loadTable->trx\_current\_load = 15.0;

break;

}

case -3:

{

loadTable->trx\_current\_load = 14.0;

break;

}

case -5:

{

loadTable->trx\_current\_load = 13.0;

break;

}

case -7:

{

loadTable->trx\_current\_load = 12.0;

break;

}

case -10:

{

loadTable->trx\_current\_load = 11.0;

break;

}

case -15:

{

loadTable->trx\_current\_load = 8.8f;

break;

}

case -25:

{

break;

}

default:

{

loadTable->trx\_current\_load =

(float)((txPower\_mW -0.1) \* 5.56 + 11.0);

break;

}

}//switch( (int)txPower\_dBm )

loadTable->sleep\_current\_load = 0.0;

loadTable->idle\_current\_load = (float)10.79/3;//mA

loadTable->rcv\_current\_load = (float)56.5/3;//mA

loadTable->voltage = 3.0;

}else if (!strcmp(str, "USER-DEFINED")){

thisPhy->eType = USER\_DEFINED\_ENERGY\_MODEL;

IO\_ReadDouble(

node->nodeId,

thisPhy->networkAddress,

nodeInput,

"ENERGY-TX-CURRENT-LOAD",

&found,

&load);

if (!found) {

loadTable->trx\_current\_load = DEFAULT\_TRX\_CURRENT\_LOAD;

} else {

loadTable->trx\_current\_load = (float ) load;

}

IO\_ReadDouble(

node->nodeId,

thisPhy->networkAddress,

nodeInput,

"ENERGY-RX-CURRENT-LOAD",

&found,

&load);

if (!found) {

loadTable->rcv\_current\_load = DEFAULT\_RCV\_CURRENT\_LOAD;

} else {

loadTable->rcv\_current\_load = (float ) load;

}

IO\_ReadDouble(

node->nodeId,

thisPhy->networkAddress,

nodeInput,

"ENERGY-IDLE-CURRENT-LOAD",

&found,

&load);

if (!found) {

loadTable->idle\_current\_load = DEFAULT\_IDLE\_CURRENT\_LOAD;

} else {

loadTable->idle\_current\_load = (float ) load;

}

IO\_ReadDouble(

node->nodeId,

thisPhy->networkAddress,

nodeInput,

"ENERGY-SLEEP-CURRENT-LOAD",

&found,

&load);

if (!found) {

loadTable->sleep\_current\_load = DEFAULT\_SLEEP\_CURRENT\_LOAD;

} else {

loadTable->sleep\_current\_load =(float ) load;

}

IO\_ReadDouble(

node->nodeId,

thisPhy->networkAddress,

nodeInput,

"ENERGY-OPERATIONAL-VOLTAGE",

&found,

&load);

if (!found) {

loadTable->voltage = DEFAULT\_OPT\_VOLTAGE;

} else {

loadTable->voltage =(float)load;

}

} else if (!strcmp(str, "GENERIC")) {

//Generic Energy Model

thisPhy->eType = GENERIC\_ENERGY\_MODEL;

IO\_ReadDouble(

node->nodeId,

thisPhy->networkAddress,

nodeInput,

"ENERGY-POWER-AMPLIFIER-INEFFICIENCY-FACTOR",

&found,

&load);

if (!found) {

thisPhy->genericEnergyModelParameters.alpha\_amp = DEFAULT\_ALPHA\_AMP;

} else {

thisPhy->genericEnergyModelParameters.alpha\_amp = (float ) load;

}

IO\_ReadDouble(

node->nodeId,

thisPhy->networkAddress,

nodeInput,

"ENERGY-TRANSMIT-CIRCUITRY-POWER-CONSUMPTION",

&found,

&load);

if (!found) {

thisPhy->genericEnergyModelParameters.Pct = DEFAULT\_PCT;

} else {

thisPhy->genericEnergyModelParameters.Pct = (float ) load;

}

IO\_ReadDouble(

node->nodeId,

thisPhy->networkAddress,

nodeInput,

"ENERGY-RECEIVE-CIRCUITRY-POWER-CONSUMPTION",

&found,

&load);

if (!found) {

thisPhy->genericEnergyModelParameters.Pcr = DEFAULT\_PCR;

} else {

thisPhy->genericEnergyModelParameters.Pcr = (float ) load;

}

IO\_ReadDouble(

node->nodeId,

thisPhy->networkAddress,

nodeInput,

"ENERGY-SLEEP-CIRCUITRY-POWER-CONSUMPTION",

&found,

&load);

if (!found) {

thisPhy->genericEnergyModelParameters.Psp = DEFAULT\_PSP;

} else {

thisPhy->genericEnergyModelParameters.Psp = (float)load;

}

IO\_ReadDouble(

node->nodeId,

thisPhy->networkAddress,

nodeInput,

"ENERGY-IDLE-CIRCUITRY-POWER-CONSUMPTION",

&found,

&load);

if (!found) {

thisPhy->genericEnergyModelParameters.Pid = DEFAULT\_PID;

} else {

thisPhy->genericEnergyModelParameters.Pid = (float)load;

}

IO\_ReadDouble(

node->nodeId,

thisPhy->networkAddress,

nodeInput,

"ENERGY-SUPPLY-VOLTAGE",

&found,

&load);

if (!found) {

thisPhy->genericEnergyModelParameters.Vs = DEFAULT\_VS;

} else {

thisPhy->genericEnergyModelParameters.Vs = (float)load;

}

thisPhy->powerConsmpTable = loadTable;

thisPhy->powerConsmpTable->voltage =

(float)thisPhy->genericEnergyModelParameters.Vs;

Generic\_UpdateCurrentLoad(node, phyIndex);

thisPhy->curLoad->load = thisPhy->powerConsmpTable->idle\_current\_load;

} else {

ERROR\_ReportError("Unknown ENERGY-MODEL-SPECIFICATION type.");

}

thisPhy->powerConsmpTable = loadTable;

thisPhy->curLoad->load =

thisPhy->powerConsmpTable->idle\_current\_load;

if (node->guiOption)

{

thisPhy->curLoad->RuntimeId =

GUI\_DefineMetric(

"Energy Model: Electrical Load (mA)",

node->nodeId,

GUI\_PHY\_LAYER,

phyIndex,

GUI\_DOUBLE\_TYPE,

GUI\_CUMULATIVE\_METRIC);

}

}

// FUNCTION: Generic\_UpdateCurrentLoad

// LAYER: PHYSICAL

// PURPOSE: To update the current load of generic energy model

// PARAMETERS:

// +node: Node\*: The node received message

// +phyIndex: index of the interface running this PHY layer

// RETURN: None

void

Generic\_UpdateCurrentLoad(

Node \*node,

const int phyIndex)

{

PhyData\* thisPhy = node->phyData[phyIndex];

double txPower\_dBm,txPower\_mW;

PHY\_GetTransmitPower(

node,

phyIndex,

&txPower\_mW);

int numConfigAntennas = PHY\_GetNumConfigAntennas(node, phyIndex);

txPower\_mW = txPower\_mW/numConfigAntennas;

txPower\_dBm =(double) 10.0 \* ( log(txPower\_mW) / log(10.0) );

thisPhy->powerConsmpTable->trx\_current\_load =

(float)((thisPhy->genericEnergyModelParameters.alpha\_amp \* txPower\_mW)

+ (thisPhy->genericEnergyModelParameters.Pct /

thisPhy->genericEnergyModelParameters.Vs));

thisPhy->powerConsmpTable->rcv\_current\_load =

(float)(thisPhy->genericEnergyModelParameters.Pcr / thisPhy->genericEnergyModelParameters.Vs);

thisPhy->powerConsmpTable->idle\_current\_load =

(float)(thisPhy->genericEnergyModelParameters.Pid / thisPhy->genericEnergyModelParameters.Vs);

thisPhy->powerConsmpTable->sleep\_current\_load =

(float)(thisPhy->genericEnergyModelParameters.Psp / thisPhy->genericEnergyModelParameters.Vs);

}

// FUNCTION: ENERGY\_PrintStats

// LAYER: PHYSICAL

// PURPOSE: To print the statistic of Energy Model

// PARAMETERS:

// +node: Node\*: The node received message

// +phyIndex: index of the interface running this PHY layer

// RETURN: None

void

ENERGY\_PrintStats(

Node \*node,

const int phyIndex)

{

PhyData\* thisPhy ;

char buf[MAX\_STRING\_LENGTH];

float volt;

double now,duration;

thisPhy = node->phyData[phyIndex];

if ((thisPhy->eType != NO\_ENERGY\_MODEL )&&

(thisPhy->energyStats))

{

volt = thisPhy->powerConsmpTable->voltage;

sprintf(buf, "Energy consumed (in mWh)in Transmit mode = %.6f",

(double)((thisPhy->curLoad->powStats.totalTxPower \*volt) / 3600.0) );

IO\_PrintStat(

node,

"Physical",

"Energy Model",

ANY\_DEST,

phyIndex,

buf);

sprintf(buf,"Energy consumed (in mWh)in Receive mode = %.6f",

(double)((thisPhy->curLoad->powStats.totalRxPower\*volt) / 3600.0) );

IO\_PrintStat(

node,

"Physical",

"Energy Model",

ANY\_DEST,

phyIndex,

buf);

sprintf(buf, "Energy consumed (in mWh)in Idle mode = %.6f",

(double)((thisPhy->curLoad->powStats.totalIdlePower\*volt) / 3600.0) );

IO\_PrintStat(

node,

"Physical",

"Energy Model",

ANY\_DEST,

phyIndex,

buf);

sprintf(buf, "Energy consumed (in mWh)in Sleep mode = %.6f",

(double)((thisPhy->curLoad->powStats.totalSleepPower\*volt) / 3600.0) );

IO\_PrintStat(

node,

"Physical",

"Energy Model",

ANY\_DEST,

phyIndex,

buf);

now = (double)

((double)node->getNodeTime()/(double)SECOND);

duration = (double)

((double)thisPhy->curLoad->powStats.totalTxDuration / (double)SECOND);

sprintf(buf, "Percentage of time in Transmit mode = %f",

(double)(duration / now)\*100.0 );

IO\_PrintStat(

node,

"Physical",

"Energy Model",

ANY\_DEST,

phyIndex,

buf);

duration = (double)

((double)thisPhy->curLoad->powStats.totalRxDuration / (double)SECOND);

sprintf(buf, "Percentage of time in Receive mode = %f",

(double)(duration / now)\*100.0 );

IO\_PrintStat(

node,

"Physical",

"Energy Model",

ANY\_DEST,

phyIndex,

buf);

duration = (double)

((double)thisPhy->curLoad->powStats.totalIdleDuration / (double)SECOND);

sprintf(buf, "Percentage of time in Idle mode = %f",

(double)(duration / now) \* 100.0 );

IO\_PrintStat(

node,

"Physical",

"Energy Model",

ANY\_DEST,

phyIndex,

buf);

duration = (double)

((double)thisPhy->curLoad->powStats.totalSleepDuration / (double)SECOND);

sprintf(buf, "Percentage of time in Sleep mode = %f",

(double)(duration / now) \* 100.0 );

IO\_PrintStat(

node,

"Physical",

"Energy Model",

ANY\_DEST,

phyIndex,

buf);

}

}

**Propogation .cpp**

#if defined(WIN32) || defined(WIN64)

#define NOMINMAX

#endif /\* Windows \*/

#define PATH\_PROFILE\_LIST\_MAX 100

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

#include <string.h>

#include <math.h>

#include <limits>

//#include "acoustics.h"

#include "api.h"

#include "node.h"

#include "partition.h"

#include "random.h"

#include "terrain.h"

#include "antenna.h"

#include "prop\_itm.h"

#include "prop\_plmatrix.h"

#ifdef ADDON\_DB

#include "dbapi.h"

#endif

#ifdef ADDON\_OPAR

//#include "prop\_opar\_r3.h"

#include "prop\_opar\_r2.h"

#endif /\*ADDON\_OPAR\*/

#ifdef TIREM\_LIB

#include "prop\_tirem.h"

#endif /\*TIREM\_LIB\*/

#ifdef ALE\_ASAPS\_LIB

#include "prop\_ips.h"

#include "prop\_asaps.h"

#endif /\*ALE\_ASAPS\_LIB\*/

#ifdef ADDON\_RFPS

#include "rfps.h"

#endif /\*ADDON\_RFPS\*/

#ifdef URBAN\_LIB

#include "prop\_cost\_hata.h"

#include "prop\_cost\_wi.h"

#include "prop\_hata.h"

#include "prop\_suburban.h"

#endif // URBAN\_LIB

#ifdef AGI\_INTERFACE

#include "agi\_interface\_util.h"

#endif

#ifdef LTE\_LIB

#include "phy\_lte.h"

#ifdef LTE\_LIB\_LOG

#include "log\_lte.h"

#endif // LTE\_LIB\_LOG

#endif // LTE\_LIB

// Fix for vc9 compilation. This must follow all the include files lest

// it be overridden again.

#if (\_MSC\_VER >= 1500) //vc9

#ifdef max

#undef max

#endif

#endif

// define an epsilon value for comparisson of double values

// since internal representations of a double value may not

// be exact

#define DOUBLE\_EPSILON 0.0000001

// Macro to compare doubles given DOUBLE\_EPSILON above...

#define DOUBLE\_IS\_EQUAL(actual, val) \

(actual + DOUBLE\_EPSILON > val && actual - DOUBLE\_EPSILON < val)

#define DEBUG 1

#define DEBUG\_AREA 1

// To check whether channel names are unique.

//

//

// \param propChannel Pointer to the first channel

// \param numberOfChannels Total number of channels

//

static

void CheckChannelNames(PropChannel\* propChannel, int numberOfChannels)

{

std::set<std::string> channelNames;

std::pair<std::set<std::string>::iterator, bool> insertedPair;

std::string channelName;

int i;

ERROR\_Assert(propChannel != NULL, "Invalid propagation channel.");

for (i = 0; i < numberOfChannels; i++)

{

channelName = propChannel[i].name;

insertedPair = channelNames.insert(channelName);

if (!insertedPair.second)

{

ERROR\_ReportErrorArgs("Error: A channel with name %s is"

" already defined. Channel names must be unique.",

channelName.c\_str());

}

}

}

double PROP\_PathlossTwoRay(double distance,

double waveLength,

float txAntennaHeight,

float rxAntennaHeight)

{

double pathloss\_dB = 0.0;

double valueForPlaneEarthLoss;

double valueForFreeSpaceLoss;

valueForPlaneEarthLoss =

distance \* distance / (txAntennaHeight \* rxAntennaHeight);

valueForFreeSpaceLoss = 4.0 \* PI \* distance / waveLength;

if (valueForPlaneEarthLoss > valueForFreeSpaceLoss) {

if (valueForPlaneEarthLoss > 1.0) {

pathloss\_dB = 20.0 \* log10(valueForPlaneEarthLoss);

}

} else {

if (valueForFreeSpaceLoss > 1.0) {

pathloss\_dB = 20.0 \* log10(valueForFreeSpaceLoss);

}

}

return pathloss\_dB;

}

double PROP\_PathlossFreeSpace(double distance,

double waveLength)

{

double pathloss\_dB = 0.0;

double valueForLog = 4.0 \* PI \* distance / waveLength;

if (valueForLog > 1.0) {

pathloss\_dB = 20.0 \* log10(valueForLog);

}

return pathloss\_dB;

}

static

void Pl\_OparInitialize(

PropChannel \*propChannel,

int channelIndex,

const NodeInput \*nodeInput,

PartitionData\* partitionData);

static

void AddObstruction(

Obstruction\* obstruction,

int obstructionIndex,

const NodeInput \*nodeInput,

int coordinateSystemType);

/\*

Calculates the extra path attenuation using OPAR model, herein

distance: the distance between Tx and Rx in meter

OverlappingDistance: the distance that the signal propagates through

the building in meter

or the distance that the signal propagates through

the foliage obstruction block in meter

frequency: frequency in Hz

ObstructionType: [OBSTRUCTION\_FOLIAGE || OBSTRUCTION\_BUILDING]

\*/

double PROP\_PathlossOpar(double distance,

double OverlappingDistance,

double frequency,

ObstructionType obstructiontype)

{

double ObstructionPenetrationLoss;

double frequencyGHz;

frequencyGHz = frequency\*1.0e-9;

assert(distance >= OverlappingDistance);

if (obstructiontype == OBSTRUCTION\_FOLIAGE)

{

if (OverlappingDistance > 14.0) {

ObstructionPenetrationLoss =

1.33 \* pow(frequencyGHz, 0.284)

\* pow(OverlappingDistance, 0.588);

}

else {

ObstructionPenetrationLoss =

0.45 \* pow( frequencyGHz, 0.284) \* OverlappingDistance;

}

}

else if (obstructiontype == OBSTRUCTION\_BUILDING)

{

ObstructionPenetrationLoss =

35.0 \* pow(0.6, frequencyGHz) + 1.0 \* OverlappingDistance;

}

else {

ObstructionPenetrationLoss = 0.0;

}

return ObstructionPenetrationLoss;

}

void PROP\_CalculatePathloss(

Node\* node,

NodeId txNodeId,

NodeId rxNodeId,

int channelIndex,

double wavelength,

float txAntennaHeight,

float rxAntennaHeight,

PropPathProfile\* pathProfile,

double\* pathloss\_dB,

bool /\* forBinning \*/)

{

double txPlatformHeight;

double rxPlatformHeight;

TerrainData\* terrainData = NODE\_GetTerrainPtr(node);

PropProfile \*propProfile = node->propChannel[channelIndex].profile;

if (DEBUG) {

printf("Calculating pathloss from node %d to node %d\n",

txNodeId, rxNodeId);

}

if (pathProfile->distance == 0.)

{

\*pathloss\_dB = 0.;

return;

}

if (propProfile->propMaxDistance > 0.1 &&

pathProfile->distance > propProfile->propMaxDistance)

{

\*pathloss\_dB = NEGATIVE\_PATHLOSS\_dB;

return;

}

switch (propProfile->pathlossModel) {

case FREE\_SPACE:

{

\*pathloss\_dB = PROP\_PathlossFreeSpace(pathProfile->distance,

wavelength);

if (DEBUG\_AREA) {

printf("pathloss model is FREE-SPACE\n");

printf("Pathloss = %f\n",

\*pathloss\_dB);

}

return;

}

case TWO\_RAY:

{

txPlatformHeight = pathProfile->fromPosition.common.c3 +

txAntennaHeight;

rxPlatformHeight = pathProfile->toPosition.common.c3 +

rxAntennaHeight;

\*pathloss\_dB = PROP\_PathlossTwoRay(pathProfile->distance,

wavelength,

(float)txPlatformHeight,

(float)rxPlatformHeight);

if (DEBUG\_AREA) {

printf("pathloss model is TWO-RAY\n");

printf("Pathloss = %f\n",

\*pathloss\_dB);

}

return;

}

case PL\_MATRIX: {

\*pathloss\_dB = NEGATIVE\_PATHLOSS\_dB;

if (DEBUG\_AREA) {

printf("pathloss model is PL\_MATRIX\n");

printf("Pathloss = %f\n",

\*pathloss\_dB);

}

return;

}

#ifdef ADDON\_OPAR

case OPAR:

{

// Because OPAR uses either Free space or TIREM, it is not

// clear how to set the platform height.

txPlatformHeight = txAntennaHeight;

rxPlatformHeight = rxAntennaHeight;

\*pathloss\_dB =

PathlossOpar(&(pathProfile->fromPosition),

&(pathProfile->toPosition),

pathProfile->distance,

(double)propProfile->frequency,

txPlatformHeight,

rxPlatformHeight);

if (DEBUG\_AREA) {

printf("pathloss model is OPAR\n");

printf("Pathloss = %f\n",

\*pathloss\_dB);

}

return;

}

#endif /\*ADDON\_OPAR\*/

case ITM:

{

int numSamples;

double elevationArray[MAX\_NUM\_ELEVATION\_SAMPLES];

if (pathProfile->distance == 0.0) {

\*pathloss\_dB = 0.0;

if (DEBUG\_AREA) {

printf("pathloss model is ITM\n");

printf("Pathloss = %f\n",

\*pathloss\_dB);

}

return;

}

numSamples =

TERRAIN\_GetElevationArray(

terrainData,

&(pathProfile->fromPosition),

&(pathProfile->toPosition),

pathProfile->distance,

propProfile->elevationSamplingDistance,

elevationArray);

if (DEBUG) {

int rm;

for (rm = 0; rm < numSamples; rm++) {

printf("sample %d is %f\n", rm, elevationArray[rm]);

}

}

if (((terrainData->getCoordinateSystem() == CARTESIAN) &&

(pathProfile->fromPosition.cartesian.y >=

pathProfile->toPosition.cartesian.y)) ||

// LATLONALT

(pathProfile->fromPosition.latlonalt.latitude >=

pathProfile->toPosition.latlonalt.latitude))

{

// the MAX adjusts for nodes that for some reason are

// below ground

txPlatformHeight

= MAX(0.0, pathProfile->fromPosition.common.c3 -

elevationArray[0]) + txAntennaHeight;

rxPlatformHeight

= MAX(0.0, pathProfile->toPosition.common.c3 -

elevationArray[numSamples]) + rxAntennaHeight;

// yes, that should be numSamples, not numSamples - 1

}

else {

txPlatformHeight

= MAX(0.0, pathProfile->fromPosition.common.c3 -

elevationArray[numSamples]) + txAntennaHeight;

rxPlatformHeight

= MAX(0.0, pathProfile->toPosition.common.c3 -

elevationArray[0]) + rxAntennaHeight;

}

//if the path is line of sight using free space, otherwise using ITM

if (PROP\_IsLineOfSight(numSamples,

pathProfile->distance / (double)numSamples,

elevationArray,

txPlatformHeight,

rxPlatformHeight,

propProfile->refractivity))

{

\*pathloss\_dB = PROP\_PathlossFreeSpace(pathProfile->distance,

wavelength);

}

else

{

\*pathloss\_dB =

PathlossItm(

numSamples + 1,

pathProfile->distance / (double)numSamples,

elevationArray,

txPlatformHeight,

rxPlatformHeight,

propProfile->polarization,

propProfile->climate,

propProfile->permittivity,

propProfile->conductivity,

propProfile->frequency / 1.0e6,

propProfile->refractivity);

}

if (DEBUG) {

printf("from (%3.4f, %3.4f) to (%3.4f, %3.4f) heights %4.2f, %4.2f: ",

pathProfile->fromPosition.latlonalt.latitude,

pathProfile->fromPosition.latlonalt.longitude,

pathProfile->toPosition.latlonalt.latitude,

pathProfile->toPosition.latlonalt.longitude,

txPlatformHeight, rxPlatformHeight);

printf("ITM: %3.4f, Free Space: %3.4f, Diff: %2.4f\n",

\*pathloss\_dB,

PROP\_PathlossFreeSpace(pathProfile->distance,

wavelength),

\*pathloss\_dB -

PROP\_PathlossFreeSpace(pathProfile->distance,

wavelength));

fflush(stdout);

}

if (DEBUG\_AREA) {

printf("pathloss model is ITM\n");

printf("Pathloss = %f\n",

\*pathloss\_dB);

}

return;

}

case RFPS:

{

#ifdef ADDON\_RFPS

Coordinates coords1;

Coordinates coords2;

if (((terrainData->getCoordinateSystem() == CARTESIAN) &&

(pathProfile->fromPosition.cartesian.y >=

pathProfile->toPosition.cartesian.y)) ||

// LATLONALT

(pathProfile->fromPosition.latlonalt.latitude >=

pathProfile->toPosition.latlonalt.latitude))

{

coords1 = pathProfile->toPosition;

coords2 = pathProfile->fromPosition;

TERRAIN\_SetToGroundLevel(terrainData, &coords1);

TERRAIN\_SetToGroundLevel(terrainData, &coords2);

// the MAX adjusts for nodes that for some reason are

// below ground

txPlatformHeight

= MAX(0.0, pathProfile->fromPosition.common.c3 -

coords1.common.c3) + txAntennaHeight;

rxPlatformHeight

= MAX(0.0, pathProfile->toPosition.common.c3 -

coords2.common.c3) + rxAntennaHeight;

// yes, that should be numSamples, not numSamples - 1

}

else

{

coords1 = pathProfile->fromPosition;

coords2 = pathProfile->toPosition;

TERRAIN\_SetToGroundLevel(terrainData, &coords1);

TERRAIN\_SetToGroundLevel(terrainData, &coords2);

txPlatformHeight

= MAX(0.0, pathProfile->fromPosition.common.c3 -

coords1.common.c3) + txAntennaHeight;

rxPlatformHeight

= MAX(0.0, pathProfile->toPosition.common.c3 -

coords2.common.c3) + rxAntennaHeight;

}

\*pathloss\_dB = RFPS\_GetPathloss(node->partitionData,

pathProfile,

txPlatformHeight,

rxPlatformHeight);

#else //ADDON\_RFPS

\*pathloss\_dB = NEGATIVE\_PATHLOSS\_dB;

#endif //ADDON\_RFPS

return;

}

case TIREM:

{

#ifdef TIREM\_LIB

int j;

Int32 numSamples;

double elevationArray[MAX\_NUM\_ELEVATION\_SAMPLES];

if (pathProfile->distance == 0.0) {

\*pathloss\_dB = 0.0;

if (DEBUG\_AREA) {

printf("pathloss model is TIREM\n");

printf("Pathloss = %f\n",

\*pathloss\_dB);

}

return;

}

numSamples =

TERRAIN\_GetElevationArray(

terrainData,

&(pathProfile->fromPosition),

&(pathProfile->toPosition),

pathProfile->distance,

propProfile->elevationSamplingDistance,

elevationArray);

if (((terrainData->getCoordinateSystem() == CARTESIAN) &&

(pathProfile->fromPosition.cartesian.y >=

pathProfile->toPosition.cartesian.y)) ||

// LATLONALT

(pathProfile->fromPosition.latlonalt.latitude >=

pathProfile->toPosition.latlonalt.latitude))

{

// the MAX adjusts for nodes that for some reason are

// below ground

txPlatformHeight

= MAX(0.0, pathProfile->fromPosition.common.c3 -

elevationArray[0]) + txAntennaHeight;

rxPlatformHeight

= MAX(0.0, pathProfile->toPosition.common.c3 -

elevationArray[numSamples]) + rxAntennaHeight;

// yes, that should be numSamples, not numSamples - 1

}

else {

txPlatformHeight

= MAX(0.0, pathProfile->fromPosition.common.c3 -

elevationArray[numSamples]) + txAntennaHeight;

rxPlatformHeight

= MAX(0.0, pathProfile->toPosition.common.c3 -

elevationArray[0]) + rxAntennaHeight;

}

\*pathloss\_dB = PathlossTirem(

node,

channelIndex,

numSamples + 1,

pathProfile->distance,

elevationArray,

txPlatformHeight,

rxPlatformHeight,

propProfile->polarizationString,

propProfile->humidity,

propProfile->permittivity,

propProfile->conductivity,

propProfile->frequency / 1.0e6,

propProfile->refractivity);

if (DEBUG) {

printf("TIREM: p %f, Free Space: %f, Diff: %f [dB]\n\n",

\*pathloss\_dB,

PROP\_PathlossFreeSpace(pathProfile->distance,

wavelength),

\*pathloss\_dB -

PROP\_PathlossFreeSpace(pathProfile->distance,

wavelength));

}

#else //TIREM\_LIB

\*pathloss\_dB = NEGATIVE\_PATHLOSS\_dB;

#endif //TIREM\_LIB

if (DEBUG\_AREA) {

printf("pathloss model is TIREM\n");

printf("Pathloss = %f\n",

\*pathloss\_dB);

}

return;

}

#ifdef ALE\_ASAPS\_LIB

case ASAPS:

{

if (pathProfile->distance < 80000) {

if (pathProfile->distance == 0.0) {

\*pathloss\_dB = 0.0;

if (DEBUG\_AREA) {

printf("pathloss model is ASAPS\n");

printf("Pathloss = %f\n",

\*pathloss\_dB);

}

return;

}

pathProfile->propDelay = (clocktype)

PROP\_CalculatePropagationDelay(

pathProfile->distance,

propProfile->wavelength \* propProfile->frequency);

\*pathloss\_dB =

PROP\_PathlossFreeSpace(pathProfile->distance,wavelength);

}

else

{

\*pathloss\_dB =

PathlossAsaps(propProfile,

pathProfile->propDelay,

&(pathProfile->fromPosition),

&(pathProfile->toPosition));

}

if (DEBUG\_AREA) {

printf("pathloss model is ASAPS\n");

printf("Pathloss = %f\n",

\*pathloss\_dB);

}

return;

}

#endif /\*ALE\_ASAPS\_LIB\*/

case PL\_OPAR:

case PL\_OPAR\_PROP:

{

// PL\_OPAR uses ITM pathloss model to calculate the pathloss and then

// uses OPAR model to calculate the extra penetration attenuation due

// to obstruction

double penetrationAttenuationTotal = 0;

double penetrationAttenuation = 0;

if (pathProfile->distance == 0.0) {

\*pathloss\_dB = 0.0;

if (DEBUG\_AREA) {

printf("pathloss model is PL\_OPAR\n");

printf("Pathloss = %f\n",

\*pathloss\_dB);

}

return;

}

if (propProfile->pathlossModelPrimary == ITM)

{

int numSamples;

double elevationArray[MAX\_NUM\_ELEVATION\_SAMPLES];

numSamples =

TERRAIN\_GetElevationArray(

terrainData,

&(pathProfile->fromPosition),

&(pathProfile->toPosition),

pathProfile->distance,

propProfile->elevationSamplingDistance,

elevationArray);

if (pathProfile->fromPosition.latlonalt.latitude >=

pathProfile->toPosition.latlonalt.latitude)

{

// the MAX adjusts for nodes that for some reason are

// below ground

txPlatformHeight

= MAX(0.0, pathProfile->fromPosition.latlonalt.altitude -

elevationArray[0]) + txAntennaHeight;

rxPlatformHeight

= MAX(0.0, pathProfile->toPosition.latlonalt.altitude -

elevationArray[numSamples]) + rxAntennaHeight;

// yes, that should be numSamples, not numSamples - 1

}

else {

txPlatformHeight

= MAX(0.0, pathProfile->fromPosition.latlonalt.altitude -

elevationArray[numSamples]) + txAntennaHeight;

rxPlatformHeight

= MAX(0.0, pathProfile->toPosition.latlonalt.altitude -

elevationArray[0]) + rxAntennaHeight;

}

//if the path is line of sight using free space, otherwise using ITM

if (PROP\_IsLineOfSight(numSamples,

pathProfile->distance / (double)numSamples,

elevationArray,

txPlatformHeight,

rxPlatformHeight,

propProfile->refractivity))

{

\*pathloss\_dB = PROP\_PathlossFreeSpace(pathProfile->distance,

wavelength);

}

else

{

\*pathloss\_dB =

PathlossItm(

numSamples + 1,

pathProfile->distance / (double)numSamples,

elevationArray,

txPlatformHeight,

rxPlatformHeight,

propProfile->polarization,

propProfile->climate,

propProfile->permittivity,

propProfile->conductivity,

propProfile->frequency / 1.0e6,

propProfile->refractivity);

}

if (DEBUG) {

printf("heights %4.2f, %4.2f: ",

txPlatformHeight, rxPlatformHeight);

printf("ITM: %3.4f, Free Space: %3.4f, Diff: %2.4f\n",

\*pathloss\_dB,

PROP\_PathlossFreeSpace(pathProfile->distance,

wavelength),

\*pathloss\_dB -

PROP\_PathlossFreeSpace(pathProfile->distance,

wavelength));

fflush(stdout);

}

} // ITM

else if (propProfile->pathlossModelPrimary == FREE\_SPACE)

{

\*pathloss\_dB = PROP\_PathlossFreeSpace(pathProfile->distance,

wavelength);

}

else if (propProfile->pathlossModelPrimary == TWO\_RAY)

{

txPlatformHeight = pathProfile->fromPosition.common.c3 +

txAntennaHeight;

rxPlatformHeight = pathProfile->toPosition.common.c3 +

rxAntennaHeight;

\*pathloss\_dB = PROP\_PathlossTwoRay(pathProfile->distance,

wavelength,

(float)txPlatformHeight,

(float)rxPlatformHeight);

}

#ifdef URBAN\_LIBRARY

// If the urban library is compiled in, we'll use the actual building

// data. Otherwise we'll just use the pre-configured numObstructions.

if (propProfile->pathlossModel == PL\_OPAR)

{

// calculate the extra penetration attenuation using OPAR model

UrbanPathPropertiesPointer pathProps =

terrainData->getUrbanPathProperties(

&(pathProfile->fromPosition),

&(pathProfile->toPosition),

0.0, false, node->partitionData);

int i;

for (i = 0; i < pathProps->getNumBuildings(); i++)

{

double distance = pathProps->getDistanceThroughBuilding(i);

if (distance > 0.0)

{

penetrationAttenuationTotal +=

PROP\_PathlossOpar(pathProfile->distance,

distance,

propProfile->frequency,

OBSTRUCTION\_BUILDING);

}

} // for

\*pathloss\_dB += penetrationAttenuationTotal;

if (DEBUG\_AREA) {

printf("pathloss model is PL\_OPAR\n");

printf("Pathloss = %f\n",

\*pathloss\_dB);

}

return;

}

else

#endif // URBAN\_LIBRARY

// This is the PL\_OPAR\_PROP case.

{

// calculate the extra penetration attenuation using OPAR model

Obstruction\* obstruction;

int numObstructions = propProfile->numObstructions;

double overlappingdistance;

double obstructionDensityFactor;

int i;

for (i = 0; i < numObstructions; i++)

{

obstruction = &(propProfile->obstructions[i]);

if ((COORD\_PointWithinRange(

terrainData->getCoordinateSystem(),

&(obstruction->southwestOrLowerLeft),

&(obstruction->northeastOrUpperRight),

&(pathProfile->fromPosition)))||

(COORD\_PointWithinRange(

terrainData->getCoordinateSystem(),

&(obstruction->southwestOrLowerLeft),

&(obstruction->northeastOrUpperRight),

&(pathProfile->toPosition))))

{

obstructionDensityFactor =

obstruction->interCityObstructionDensityFactor;

if ((COORD\_PointWithinRange(

terrainData->getCoordinateSystem(),

&(obstruction->southwestOrLowerLeft),

&(obstruction->northeastOrUpperRight),

&(pathProfile->fromPosition)))&&

(COORD\_PointWithinRange(

terrainData->getCoordinateSystem(),

&(obstruction->southwestOrLowerLeft),

&(obstruction->northeastOrUpperRight),

&(pathProfile->toPosition))))

{

obstructionDensityFactor =

obstruction->intraCityObstructionDensityFactor;

}

if ((int)pathProfile->distance == 0)

{

overlappingdistance = 0;

}

else

{

// coordinate system is still being overhauled

// not all coordinates have been updated

// to have type, yet

int coordinateSystemType =

NODE\_GetTerrainPtr(node)->getCoordinateSystem();

if (coordinateSystemType == LATLONALT)

{

pathProfile->fromPosition.type = GEODETIC;

pathProfile->toPosition.type = GEODETIC;

}

else

{

pathProfile->fromPosition.type = UNREFERENCED\_CARTESIAN;

pathProfile->toPosition.type = UNREFERENCED\_CARTESIAN;

}

RandomSeed oparSeed;

// We want the random number to change over time, but be the same for a node

// pair over a period of time, so we use the node IDs and the most significant digits

// of the current time to create the seed.

RANDOM\_SetSeed(oparSeed,

MIN(txNodeId, rxNodeId),

MAX(txNodeId, rxNodeId),

(int) (node->getNodeTime() >> 8));

overlappingdistance =

(double)((int)RANDOM\_nrand(oparSeed) %

(int)pathProfile->distance) \* obstructionDensityFactor;

}

if (overlappingdistance > 0.0)

{

penetrationAttenuation =

PROP\_PathlossOpar(pathProfile->distance,

overlappingdistance,

propProfile->frequency,

obstruction->obstructiontype);

if (DEBUG) {

char buf[MAX\_STRING\_LENGTH];

TIME\_PrintClockInSecond(node->getNodeTime(), buf);

printf("distance %4.2f overlapping distance %4.2f"

" obstruction loss %4.2f simtime %sS\n",

pathProfile->distance,

overlappingdistance,

penetrationAttenuation,

buf);

fflush(stdout);

}

}

else

{

penetrationAttenuation = 0.0;

}

} // if

else

{

penetrationAttenuation = 0.0;

}

penetrationAttenuationTotal += penetrationAttenuation;

} // for

\*pathloss\_dB += penetrationAttenuationTotal;

if (DEBUG\_AREA) {

printf("pathloss model is PL\_OPAR\_PROP\n");

printf("Pathloss = %f\n",

\*pathloss\_dB);

}

return;

}

}

#ifdef URBAN\_LIB

case OKUMURA\_HATA:

{

\*pathloss\_dB =

PathlossHata(pathProfile->distance,

wavelength,

txAntennaHeight,

rxAntennaHeight,

propProfile);

if (DEBUG) {

printf("OKUMURA\_HATA: %f, Free Space: %f, Diff: %f [dB]\n\n",

\*pathloss\_dB,

PROP\_PathlossFreeSpace(pathProfile->distance,

wavelength),

\*pathloss\_dB -

PROP\_PathlossFreeSpace(pathProfile->distance,

wavelength));

}

return;

}

case COST231\_HATA:

{

\*pathloss\_dB =

PathlossCOST231Hata(pathProfile->distance,

wavelength,

txAntennaHeight,

rxAntennaHeight,

propProfile);

if (DEBUG) {

printf("COST231\_HATA: %f, Free Space: %f, Diff: %f [dB]\n\n",

\*pathloss\_dB,

PROP\_PathlossFreeSpace(pathProfile->distance,

wavelength),

\*pathloss\_dB -

PROP\_PathlossFreeSpace(pathProfile->distance,

wavelength));

}

return;

}

case COST231\_WALFISH\_IKEGAMI:

{

\*pathloss\_dB =

PathlossCOST231\_WI(node,

pathProfile->distance,

wavelength,

txAntennaHeight,

rxAntennaHeight,

propProfile);

if (DEBUG) {

printf("COST231\_WI: %f, Free Space: %f, Diff: %f [dB]\n\n",

\*pathloss\_dB,

PROP\_PathlossFreeSpace(pathProfile->distance,

wavelength),

\*pathloss\_dB -

PROP\_PathlossFreeSpace(pathProfile->distance,

wavelength));

}

return;

}

case URBAN\_MODEL\_AUTOSELECT:

{

\*pathloss\_dB = Pathloss\_UrbanProp(node,

txNodeId,

rxNodeId,

wavelength,

txAntennaHeight,

rxAntennaHeight,

propProfile,

pathProfile);

if (DEBUG) {

printf("URBAN AUTOSELECT: %f, Free Space: %f, Diff: %f [dB]\n\n",

\*pathloss\_dB, PROP\_PathlossFreeSpace(pathProfile->distance,

wavelength),

\*pathloss\_dB - PROP\_PathlossFreeSpace(pathProfile->distance,

wavelength));

}

return;

}

case STREET\_M\_TO\_M:

{

\*pathloss\_dB = Pathloss\_Street\_M\_to\_M(

propProfile->Num\_builings\_in\_path,

txAntennaHeight,

rxAntennaHeight,

propProfile->roofHeight,

propProfile->streetWidth,

pathProfile->distance,

wavelength);

if (DEBUG) {

printf("COST231\_HATA: %f, Free Space: %f, Diff: %f [dB]\n\n",

\*pathloss\_dB,

PROP\_PathlossFreeSpace(pathProfile->distance,

wavelength),

\*pathloss\_dB -

PROP\_PathlossFreeSpace(pathProfile->distance,

wavelength));

}

return;

}

case STREET\_MICROCELL:

{

double txDistanceToBuilding = pathProfile->distance/2.;

double rxDistanceToBuilding = pathProfile->distance/2.;

double distanceThruBuilding = 0;

if (propProfile->losIndicator == NLOS)

\*pathloss\_dB = Pathloss\_StreetMicrocell\_NLoS(txAntennaHeight,

rxAntennaHeight,

wavelength,

//propProfile->frequency,

txDistanceToBuilding,

rxDistanceToBuilding,

distanceThruBuilding);

else

\*pathloss\_dB = Pathloss\_StreetMicrocell\_LoS(txAntennaHeight,

rxAntennaHeight,

wavelength,

//propProfile->frequency,

pathProfile->distance);

if (DEBUG) {

printf("STREET-MICROCELL: %f, Free Space: %f, Diff: %f [dB]\n\n",

\*pathloss\_dB, PROP\_PathlossFreeSpace(pathProfile->distance,

wavelength),

\*pathloss\_dB - PROP\_PathlossFreeSpace(pathProfile->distance,

wavelength));

}

return;

}

case SUBURBAN\_FOLIAGE:

{

\*pathloss\_dB = PROP\_Pathloss\_Suburban(

(double)propProfile->frequency / 1.0e6, //in MHz

wavelength,

pathProfile->distance,

txAntennaHeight,

rxAntennaHeight,

propProfile->suburbanTerrainType);

if (DEBUG) {

printf("SUBURBAN: %f, Free Space: %f, Diff: %f [dB]\n\n",

\*pathloss\_dB, PROP\_PathlossFreeSpace(pathProfile->distance,

wavelength),

\*pathloss\_dB - PROP\_PathlossFreeSpace(pathProfile->distance,

wavelength));

}

return;

}

#endif // URBAN\_LIB

default: {

ERROR\_ReportError("Invalid pathloss model selected");

}

}

return;

}

//

// RandomizeGaussianComponentStartingPoint() returns an integer in [0, arraySize)

// It is to be used for determining the starting point of the fading data array

// In order for both sides of a link to have the same fading effect at the same

// simulation time, nodeId1 and nodeId2 are sorted before configuring the seed

//

static

int RandomizeGaussianComponentStartingPoint(

NodeAddress nodeId1,

NodeAddress nodeId2,

int channelIndex,

int arraySize)

{

RandomSeed seed;

RANDOM\_SetSeed(seed,

MIN(nodeId1,nodeId2),

MAX(nodeId1,nodeId2),

channelIndex);

return RANDOM\_nrand(seed) % arraySize;

}

// assuming here that the receiving node (node 2) is always local, while transmitter might be remote.

// also assuming that fading stretching factor is the same for both nodes

void PROP\_CalculateFading(

Message\* signalMsg,

PropTxInfo\* propTxInfo,

Node\* node2,

int channelIndex,

clocktype currentTime,

float\* fading\_dB,

double\* /\* channelReal \*/,

double\* /\* channelImag \*/)

{

PropChannel\* propChannel = node2->partitionData->propChannel;

PropProfile\* propProfile = propChannel[channelIndex].profile;

PropProfile\* propProfile0 = propChannel[0].profile;

#ifdef LTE\_LIB

// LTE library bypasses regular fading calculation here. It is done

// for each OFDMA TransportBlock to simulate MIMO channel in PHY.

PhyLteTxInfo\* lteTxInfo = (PhyLteTxInfo\*) MESSAGE\_ReturnInfo(

signalMsg,

INFO\_TYPE\_LtePhyTxInfo);

if ((propProfile->fadingModel == RICEAN) && (lteTxInfo != NULL)) {

\*fading\_dB = 0.0;

return;

}

#endif // LTE\_LIB

if (propProfile->fadingModel == RICEAN) {

int arrayIndex;

double arrayIndexInDouble;

double value1, value2;

const float kFactor = (float)propProfile->kFactor;

const int numGaussianComponents = propProfile0->numGaussianComponents;

const int startingPoint =

RandomizeGaussianComponentStartingPoint(

propTxInfo->txNodeId, node2->nodeId, channelIndex,

numGaussianComponents);

if (propProfile->motionEffectsEnabled){

PROP\_MotionObtainfadingStretchingFactor(propTxInfo,

node2,

channelIndex);

}

arrayIndexInDouble =

node2->propData[channelIndex].fadingStretchingFactor \*

(double)currentTime;

arrayIndexInDouble -=

(double)numGaussianComponents \*

floor(arrayIndexInDouble / (double)numGaussianComponents);

arrayIndex =

(RoundToInt(arrayIndexInDouble) + startingPoint) %

numGaussianComponents;

value1 = propProfile0->gaussianComponent1[arrayIndex] +

sqrt(2.0 \* kFactor);

value2 = propProfile0->gaussianComponent2[arrayIndex];

\*fading\_dB =

(float)IN\_DB((value1 \* value1 + value2 \* value2) / (2.0 \* (kFactor + 1)));

}

else {

\*fading\_dB = 0.0;

}

}

// Returns true if shadowing applies to the pathloss calculation.

bool PROP\_ShadowingApplies(

Node \*node,

int channelIndex)

{

PropProfile\* propProfile = node->propChannel[channelIndex].profile;

if (propProfile->pathlossModel == FREE\_SPACE)

return true;

else if (propProfile->pathlossModel == TWO\_RAY)

return true;

else if (propProfile->pathlossModel == PL\_OPAR ||

propProfile->pathlossModel == PL\_OPAR\_PROP)

{

if (propProfile->pathlossModelPrimary == FREE\_SPACE ||

propProfile->pathlossModelPrimary == TWO\_RAY)

return true;

}

return false;

}

// Returns true and calculates shadowing when it applies.

// Returns false and sets shadowing\_dB to zero otherwise.

bool PROP\_CalculateShadowing(

Node\* node,

int channelIndex,

double\* shadowing\_dB)

{

if (PROP\_ShadowingApplies(node, channelIndex))

{

PropData\* propData = &(node->propData[channelIndex]);

// getRandomNumber also handles constant shadowing

\*shadowing\_dB = propData->shadowingDistribution.getRandomNumber();

return true;

}

\*shadowing\_dB = 0.0;

return false;

}

// This function will be called by QualNet wireless

// propagation code to calculate rxPower and prop delay

// for a specific signal from a specific tx node to

// a specific rx node.

//

// \param msg Signal to be propagated

// \param channelIndex Channel that the signal is propagated

// \param propChannel Info of the propagation channel

// \param propTxInfo Transmission parameers of the tx node

// \param txNode Point to the Tx node

// \param rxNode Point to the Rx node

// \param pathProfile For returning results

//

// \return If FALSE, indicate the two nodes cannot comm

// TRUE means two nodes can communicate

BOOL PROP\_CalculateRxPowerAndPropagationDelay(

Message\* msg,

int channelIndex,

PropChannel\* propChannel,

PropTxInfo\* propTxInfo,

Node\* txNode,

Node\* rxNode,

PropPathProfile\* pathProfile)

{

#ifdef AGI\_INTERFACE

if (txNode->partitionData->isAgiInterfaceEnabled)

{

if (txNode->nodeId == rxNode->nodeId)

return FALSE;

PhyData\* txPhy = txNode->phyData[propTxInfo->phyIndex];

PhyData\* rxPhy = rxNode->phyData[0]; // use the first PHY on the rx-er

assert(rxPhy != NULL); // make sure there is a first phy

NodeInterfacePair xmtrId(txNode->nodeId, txPhy->macInterfaceIndex);

NodeInterfacePair rcvrId(rxNode->nodeId, rxPhy->macInterfaceIndex);

CAgiInterfaceUtil::ComputeRequest req;

req.time\_nanoseconds = txNode->getNodeTime();

req.xmtrId = xmtrId;

req.rcvrId = rcvrId;

req.channelIndex = channelIndex;

req.frequency\_hertz = propChannel->profile->frequency;

req.power\_dBm = propTxInfo->txPower\_dBm;

req.dataRate\_bps = propTxInfo->dataRate;

return CAgiInterfaceUtil::GetInstance().ComputeLink(req, pathProfile);

}

else

#endif

{

return PROP\_DefaultCalculateRxPowerAndPropagationDelay(

msg,

channelIndex,

propChannel,

propTxInfo,

txNode,

rxNode,

pathProfile);

}

}

/\*

\* FUNCTION PROP\_GlobalInit

\* PURPOSE Initialization function for propagation models

\* This function is called from each partition, not from each node

\*

\* Parameters:

\* propData: structure shared among nodes

\* nodeInput: structure containing contents of input file

\*/

void PROP\_GlobalInit(PartitionData \*partitionData, NodeInput \*nodeInput) {

BOOL wasFound;

char buf[MAX\_STRING\_LENGTH];

PropChannel\* propChannel;

PropProfile\* propProfile;

PropProfile\* propProfile0 = NULL; // propChannel[0].profile

TerrainData\* terrainData = PARTITION\_GetTerrainPtr(partitionData);

double frequency;

double height;

double propLimit\_dB;

double propSpeed;

Float64 propMaxDistance;

double propCommunicationProximity;

double propProfileUpdateRatio;

int channelIndex = 0;

int profileIndex = 0;

int numChannels = 0;

int numFixedChannels = 0;

int i;

double shadowingMean\_dB;

double kFactor;

double maxVelocity;

BOOL fadingOnAnyChannel = FALSE;

std::string name;

char idStr[MAX\_STRING\_LENGTH];

//

// Scan how many channels are defined

//

while (TRUE) {

IO\_ReadStringInstance(

ANY\_NODEID,

ANY\_ADDRESS,

nodeInput,

"PROPAGATION-CHANNEL-FREQUENCY",

numChannels,

(numChannels == 0),

&wasFound,

buf);

if (!wasFound) {

break;

}

numChannels++;

}

numFixedChannels = numChannels;

if (numChannels == 0) {

// no channel is defined

partitionData->propChannel = NULL;

partitionData->numChannels = 0;

partitionData->numFixedChannels = 0;

partitionData->numProfiles = 0;

return;

}

else

{

// These should be set earlier rather than end of the function as

// they used in some other initialize functions.

partitionData->numChannels = numChannels;

partitionData->numFixedChannels = numFixedChannels;

}

propChannel = new PropChannel[numChannels];

partitionData->propChannel = propChannel;

for (i = 0; i < numChannels; i++) {

//

// Get the channel frequency

//

IO\_ReadStringInstance(

ANY\_NODEID,

ANY\_ADDRESS,

nodeInput,

"PROPAGATION-CHANNEL-FREQUENCY",

channelIndex,

(channelIndex == 0),

&wasFound,

buf);

assert(wasFound == TRUE);

if (strncmp(buf, "SAME-AS-", 8) == 0) {

int channelIndexToReferTo = (int)atoi(&buf[8]);

if (channelIndexToReferTo >= channelIndex) {

ERROR\_ReportError(

"ERROR: 'SAME-AS-\*' keyword for "

"PROPAGATION-CHANNEL-FREQUENCY works\n"

"only when the referred channel index "

"is smaller than itself\n");

}

propChannel[channelIndex].profileIndex =

propChannel[channelIndexToReferTo].profileIndex;

propChannel[channelIndex].profile =

propChannel[channelIndexToReferTo].profile;

assert(propChannel[channelIndex].profileIndex ==

propChannel[channelIndex].profile->profileIndex);

propChannel[channelIndex].numNodes = 0;

propChannel[channelIndex].nodeList = NULL;

propChannel[channelIndex].numNodesWithLI = 0;

propChannel[channelIndex].nodeListWithLI = NULL;

sprintf(idStr, "channel%d", channelIndex);

propChannel[channelIndex].name = idStr;

channelIndex++;

continue;

}

propChannel[channelIndex].profileIndex = profileIndex;

propChannel[channelIndex].profile = new PropProfile;

propProfile = propChannel[channelIndex].profile;

propProfile->profileIndex = profileIndex;

if (channelIndex == 0) {

propProfile->numChannelsInMatrix = 0;

}

propChannel[channelIndex].numNodes = 0;

propChannel[channelIndex].nodeList = NULL;

propChannel[channelIndex].numNodesWithLI = 0;

propChannel[channelIndex].nodeListWithLI = NULL;

frequency = (double)atof(buf);

propProfile->propGlobalVar = NULL;

propProfile->frequency = frequency;

propProfile->motionEffectsEnabled = FALSE;

// Get the channel name

IO\_ReadStringInstance(

ANY\_NODEID,

ANY\_ADDRESS,

nodeInput,

"PROPAGATION-CHANNEL-NAME",

channelIndex,

(channelIndex == 0),

&wasFound,

buf);

if (!wasFound) {

sprintf(idStr, "channel%d", channelIndex);

name = idStr;

}

else

{

if (isalpha(\*buf) &&

(strchr(buf, ' ') == NULL) &&

(strchr(buf, ',') == NULL) &&

(strchr(buf, '[') == NULL) &&

(strchr(buf, ']') == NULL) &&

(strchr(buf, '{') == NULL) &&

(strchr(buf, '}') == NULL) &&

(strchr(buf, '(') == NULL) &&

(strchr(buf, ')') == NULL))

{

name = buf;

}

else

{

ERROR\_ReportErrorArgs(

"PROPAGATION-CHANNEL-NAME[%d] has an invalid channel"

" name. Channel name can have any characters except "

"\"comma, space, (, ), {, }, [ and ].\" Channel "

"name should begin with a lowercase or uppercase "

"alphabet. Channel name cannot be left blank.",

channelIndex);

}

}

propChannel[channelIndex].name = name;

//Default height for the channel frequency

IO\_ReadStringInstance(

ANY\_NODEID,

ANY\_ADDRESS,

nodeInput,

"ANTENNA-HEIGHT",

channelIndex,

(channelIndex == 0),

&wasFound,

buf);

if (!wasFound)

height = ANTENNA\_DEFAULT\_HEIGHT;

else

height = (double)atof(buf);

propProfile->antennaHeight = height; //default height for the prop. frequency

//

// Get the signal propagation speed.

//

IO\_ReadDoubleInstance(

ANY\_NODEID,

ANY\_ADDRESS,

nodeInput,

"PROPAGATION-SPEED",

channelIndex,

TRUE,

&wasFound,

&propSpeed);

if (!wasFound) {

propSpeed = SPEED\_OF\_LIGHT;

}

propProfile->wavelength = propSpeed / frequency;

//

// Get the propagation limit.

//

IO\_ReadDoubleInstance(

ANY\_NODEID,

ANY\_ADDRESS,

nodeInput,

"PROPAGATION-LIMIT",

channelIndex,

TRUE,

&wasFound,

&propLimit\_dB);

if (wasFound) {

propProfile->propLimit\_dB = propLimit\_dB;

}

else {

propProfile->propLimit\_dB = PROP\_DEFAULT\_PROPAGATION\_LIMIT\_dBm;

}

//

// Get the propagation limit.

//

IO\_ReadDoubleInstance(

ANY\_NODEID,

ANY\_ADDRESS,

nodeInput,

"PROPAGATION-MAX-DISTANCE",

channelIndex,

TRUE,

&wasFound,

&propMaxDistance);

if (wasFound) {

propProfile->propMaxDistance = propMaxDistance;

}

else {

propProfile->propMaxDistance = 0.0;

}

//

// Get the propagation communication proximity

// where path profile is updated for every position change.

//

// Get the Propagation proximity only if external Interface is turned off

IO\_ReadDoubleInstance(

ANY\_NODEID,

ANY\_ADDRESS,

nodeInput,

"PROPAGATION-COMMUNICATION-PROXIMITY",

channelIndex,

TRUE,

&wasFound,

&propCommunicationProximity);

if (wasFound)

{

propProfile->propCommunicationProximity = propCommunicationProximity;

}

else {

propProfile->propCommunicationProximity = 400.0;

}

//

// Get the path profile update ratio.

//

// This change the frequency to trigger a path profile update

// for a pair of nodes not in their proximity. For instance,

// the path profile for two nodes that are D m away from each

// other is updated if D is changed as much as:

//

// (D - PROPAGATION-PROXIMITY-DISTANCE) \*

// (PROPAGATION-PROFILE-UPDATE-RATIO).

//

IO\_ReadDoubleInstance(

ANY\_NODEID,

ANY\_ADDRESS,

nodeInput,

"PROPAGATION-PROFILE-UPDATE-RATIO",

channelIndex,

TRUE,

&wasFound,

&propProfileUpdateRatio);

if (wasFound) {

propProfile->propProfileUpdateRatio = propProfileUpdateRatio;

}

else {

propProfile->propProfileUpdateRatio = 0.0;

}

propProfile0 = propChannel[0].profile;

assert(propProfile0 != NULL);

BOOL wasEnabled = FALSE;

IO\_ReadBoolInstance(

ANY\_NODEID,

ANY\_ADDRESS,

nodeInput,

"PROPAGATION-ENABLE-CHANNEL-OVERLAP-CHECK",

channelIndex,

TRUE,

&wasFound,

&wasEnabled);

if (wasFound && wasEnabled) {

propProfile->enableChannelOverlapCheck = TRUE;

}

else {

propProfile->enableChannelOverlapCheck = FALSE;

}

//

// Set pathlossModel

//

IO\_ReadStringInstance(

ANY\_NODEID,

ANY\_ADDRESS,

nodeInput,

"PROPAGATION-PATHLOSS-MODEL",

channelIndex,

TRUE,

&wasFound,

buf);

if (wasFound) {

if (strcmp(buf, "FREE-SPACE") == 0) {

propProfile->pathlossModel = FREE\_SPACE;

}

else if (strcmp(buf, "TWO-RAY") == 0) {

propProfile->pathlossModel = TWO\_RAY;

}

else if (strcmp(buf, "PATHLOSS-MATRIX") == 0) {

propProfile->pathlossModel = PL\_MATRIX;

propProfile0->numChannelsInMatrix++;

}

else if (strcmp(buf, "OPAR") == 0) {

#ifdef ADDON\_OPAR

propProfile->pathlossModel = OPAR;

if (terrainData->getCoordinateSystem() !=

LATLONALT)

{

ERROR\_ReportError(

"OPAR requires LATLONALT coordinate system type");

}

OparInitialize(

&(propChannel[channelIndex]), channelIndex, nodeInput);

#else /\*ADDON\_OPAR\*/

ERROR\_ReportMissingAddon(buf, "OPAR");

#endif /\*ADDON\_OPAR\*/

}

else if (strcmp(buf, "ITM") == 0) {

propProfile->pathlossModel = ITM;

if (!terrainData->hasElevationData()) {

ERROR\_ReportError("ITM requires terrain data\n");

}

ItmInitialize(

&(propChannel[channelIndex]), channelIndex, nodeInput);

}

else if (strcmp(buf, "TIREM") == 0) {

propProfile->pathlossModel = TIREM;

if (!terrainData->hasElevationData()) {

ERROR\_ReportError("TIREM requires terrain data");

}

#ifdef TIREM\_LIB

TiremInitialize(

&(propChannel[channelIndex]), channelIndex, nodeInput);

#else //TIREM\_LIB

ERROR\_ReportMissingLibrary(buf, "TIREM");

#endif //TIREM\_LIB

}

else if (strcmp(buf, "ASAPS") == 0) {

#ifdef ALE\_ASAPS\_LIB

propProfile->pathlossModel = ASAPS;

if (terrainData->getCoordinateSystem() !=

LATLONALT)

{

ERROR\_ReportError(

"ASAPS requires LATLONALT coordinate system type\n");

}

Prop\_AsapsInitialize(

&(propChannel[channelIndex]), channelIndex, nodeInput);

#else //ALE\_ASAPS\_LIB

ERROR\_ReportMissingLibrary(buf, "ALE/ASAPS");

#endif //ALE\_ASAPS\_LIB

}

else if (strcmp(buf, "PATHLOSS-OPAR") == 0) {

propProfile->pathlossModel = PL\_OPAR;

IO\_ReadStringInstance(

ANY\_NODEID,

ANY\_ADDRESS,

nodeInput,

"PROPAGATION-PATHLOSS-MODEL-PRIMARY",

channelIndex,

TRUE,

&wasFound,

buf);

if (!wasFound || (strcmp(buf, "ITM") == 0))

{

if (!terrainData->hasElevationData()) {

ERROR\_ReportError("ITM requires terrain data\n");

}

propProfile->pathlossModelPrimary = ITM;

ItmInitialize(

&(propChannel[channelIndex]),

channelIndex,

nodeInput);

}

else if (strcmp(buf, "TWO-RAY") == 0)

{

propProfile->pathlossModelPrimary = TWO\_RAY;

}

else if (strcmp(buf, "FREE-SPACE") == 0)

{

propProfile->pathlossModelPrimary = FREE\_SPACE;

}

else

{

char errorStr[MAX\_STRING\_LENGTH];

sprintf(errorStr,

"Error: unknown PROPAGATION-PATHLOSS-MODEL-PRIMARY '%s'.\n",

buf);

ERROR\_ReportError(errorStr);

}

}

else if (strcmp(buf, "PATHLOSS-OPAR-PROP") == 0) {

propProfile->pathlossModel = PL\_OPAR\_PROP;

IO\_ReadStringInstance(

ANY\_NODEID,

ANY\_ADDRESS,

nodeInput,

"PROPAGATION-PATHLOSS-MODEL-PRIMARY",

channelIndex,

TRUE,

&wasFound,

buf);

if (!wasFound || (strcmp(buf, "ITM") == 0))

{

if (!terrainData->hasElevationData()) {

ERROR\_ReportError("ITM requires terrain data\n");

}

propProfile->pathlossModelPrimary = ITM;

ItmInitialize(

&(propChannel[channelIndex]),

channelIndex,

nodeInput);

}

else if (strcmp(buf, "TWO-RAY") == 0)

{

propProfile->pathlossModelPrimary = TWO\_RAY;

}

else if (strcmp(buf, "FREE-SPACE") == 0)

{

propProfile->pathlossModelPrimary = FREE\_SPACE;

}

else

{

char errorStr[MAX\_STRING\_LENGTH];

sprintf(errorStr,

"Error: unknown PROPAGATION-PATHLOSS-MODEL-PRIMARY '%s'.\n",

buf);

ERROR\_ReportError(errorStr);

}

Pl\_OparInitialize(

&(propChannel[channelIndex]),

channelIndex,

nodeInput,

partitionData);

}

else if (strcmp(buf, "RFPS") == 0) {

#ifdef ADDON\_RFPS

propProfile->pathlossModel = RFPS;

if (!terrainData->hasElevationData()) {

ERROR\_ReportError("RFPS/TIREM requires terrain data\n");

}

RFPS\_Init(partitionData,

&(propChannel[channelIndex]),

channelIndex,

nodeInput);

#else //ADDON\_RFPS

ERROR\_ReportMissingAddon(buf, "RFPS");

#endif //ADDON\_RFPS

}

else if (strcmp(buf, "OKUMURA-HATA") == 0) {

#ifdef URBAN\_LIB

propProfile->pathlossModel = OKUMURA\_HATA;

char errorStr[MAX\_STRING\_LENGTH];

double frequencyMhz = frequency \* 1.0e-6;

if (!((frequencyMhz > 150.0) &&

(frequencyMhz < 1000.0)))

{

sprintf(errorStr, "Frequency = %f MHz; Not in recommended "

"range [150:1000]MHz for use in OKUMURA-HATA\n",

frequencyMhz);

ERROR\_ReportWarning(errorStr);

}

Okumura\_HataInitialize(

&(propChannel[channelIndex]), channelIndex, nodeInput);

#else // URBAN\_LIB

ERROR\_ReportMissingLibrary(buf, "Urban");

#endif // URBAN\_LIB

}

else if (strcmp(buf, "COST231-HATA") == 0) {

#ifdef URBAN\_LIB

propProfile->pathlossModel = COST231\_HATA;

char errorStr[MAX\_STRING\_LENGTH];

double frequencyMhz = frequency \* 1.0e-6;

if (!((frequencyMhz > 1500.0) &&

(frequencyMhz < 2000.0)))

{

sprintf(errorStr, "Frequency = %f MHz; Not in recommended "

"range [1500:2000] MHz for use in COST231-HATA Model\n",

frequencyMhz);

ERROR\_ReportWarning(errorStr);

}

COST231\_HataInitialize(

&(propChannel[channelIndex]), channelIndex, nodeInput);

#else // URBAN\_LIB

ERROR\_ReportMissingLibrary(buf, "Urban");

#endif // URBAN\_LIB

}

else if (strcmp(buf, "COST231-WALFISH-IKEGAMI") == 0) {

#ifdef URBAN\_LIB

propProfile->pathlossModel = COST231\_WALFISH\_IKEGAMI;

double frequencyMhz = frequency \* 1.0e-6;

char errorStr[MAX\_STRING\_LENGTH];

if (!((frequencyMhz > 800.0) &&

(frequencyMhz <2000.0))) {

sprintf(errorStr, "Frequency = %f MHz; Not in recommended "

"range [800:2000] MHz for use in COST-Walfish Ikegami Model\n",

frequencyMhz);

ERROR\_ReportWarning(errorStr);

}

COST231\_WIInitialize(

&(propChannel[channelIndex]), channelIndex, nodeInput);

#else // URBAN\_LIB

ERROR\_ReportMissingLibrary(buf, "Urban");

#endif // URBAN\_LIB

}

else if (strcmp(buf, "URBAN-MODEL-AUTOSELECT") == 0) {

#ifdef URBAN\_LIB

propProfile->pathlossModel = URBAN\_MODEL\_AUTOSELECT;

UrbanProp\_Initialize( &(propChannel[channelIndex]),

channelIndex,

nodeInput);

#else // URBAN\_LIB

ERROR\_ReportMissingLibrary(buf, "Urban");

#endif // URBAN\_LIB

}

else if (strcmp(buf, "STREET-MICROCELL") == 0) {

#ifdef URBAN\_LIB

propProfile->pathlossModel = STREET\_MICROCELL;

StreetMicrocell\_Initialize(&(propChannel[channelIndex]),

channelIndex, nodeInput);

#else // URBAN\_LIB

ERROR\_ReportMissingLibrary(buf, "Urban");

#endif // URBAN\_LIB

}

else if (strcmp(buf, "STREET-M-TO-M") == 0) {

#ifdef URBAN\_LIB

propProfile->pathlossModel = STREET\_M\_TO\_M;

Street\_M\_to\_M\_Initialize(&(propChannel[channelIndex]),

channelIndex, nodeInput);

#else // URBAN\_LIB

ERROR\_ReportMissingLibrary(buf, "Urban");

#endif // URBAN\_LIB

}

else if (strcmp(buf, "SUBURBAN") == 0) {

#ifdef URBAN\_LIB

propProfile->pathlossModel = SUBURBAN\_FOLIAGE;

Suburban\_Initialize(&(propChannel[channelIndex]),

channelIndex, nodeInput);

#else // URBAN\_LIB

ERROR\_ReportMissingLibrary(buf, "Urban");

#endif // URBAN\_LIB

}

else {

char errorStr[MAX\_STRING\_LENGTH];

sprintf(errorStr,

"Error: unknown or unsupported PROPAGATION-PATHLOSS-MODEL '%s'.\n",

buf);

ERROR\_ReportError(errorStr);

}

}

else {

propProfile->pathlossModel = TWO\_RAY;

}

//

// Set shadowingModel

//

IO\_ReadStringInstance(

ANY\_NODEID,

ANY\_ADDRESS,

nodeInput,

"PROPAGATION-SHADOWING-MODEL",

channelIndex,

TRUE,

&wasFound,

buf);

if (wasFound) {

if (strcmp(buf, "NONE") == 0) {

propProfile->shadowingModel = CONSTANT;

propProfile->shadowingMean\_dB = 0.0;

}

else {

if (strcmp(buf, "LOGNORMAL") == 0) {

propProfile->shadowingModel = LOGNORMAL;

}

else if (strcmp(buf, "CONSTANT") == 0) {

propProfile->shadowingModel = CONSTANT;

}

else {

char errorMessage[MAX\_STRING\_LENGTH];

sprintf(errorMessage,

"Error: unknown PROPAGATION-SHADOWING-MODEL '%s'.\n",

buf);

ERROR\_ReportError(errorMessage);

}

//

// Set mean value of shadowing effect

//

IO\_ReadDoubleInstance(

ANY\_NODEID,

ANY\_ADDRESS,

nodeInput,

"PROPAGATION-SHADOWING-MEAN",

channelIndex,

TRUE,

&wasFound,

&shadowingMean\_dB);

if (wasFound) {

propProfile->shadowingMean\_dB = shadowingMean\_dB;

}

else {

propProfile->shadowingMean\_dB = PROP\_DEFAULT\_SHADOWING\_MEAN\_dB;

}

}

if ((propProfile->pathlossModel != FREE\_SPACE) &&

(propProfile->pathlossModel != TWO\_RAY) &&

((propProfile->shadowingModel != CONSTANT) ||

DOUBLE\_IS\_EQUAL(0, propProfile->shadowingMean\_dB) == FALSE))

{

char errorStr[MAX\_STRING\_LENGTH];

sprintf(errorStr, "\nNote: Only FREE SPACE and TWO RAY"

" pathloss model take shadowing model into"

" account to calculate pathloss\n\n");

ERROR\_ReportWarning(errorStr);

}

}

else {

propProfile->shadowingModel = CONSTANT;

propProfile->shadowingMean\_dB = PROP\_DEFAULT\_SHADOWING\_MEAN\_dB;

}

//

// Set fadingModel

//

IO\_ReadStringInstance(

ANY\_NODEID,

ANY\_ADDRESS,

nodeInput,

"PROPAGATION-FADING-MODEL",

channelIndex,

TRUE,

&wasFound,

buf);

if (wasFound) {

if (strcmp(buf, "NONE") == 0) {

propProfile->fadingModel = NONE;

}

else if (strcmp(buf, "RAYLEIGH") == 0) {

//

// When Rayleigh fading is specified, Ricean with K = 0 is

// actually set.

//

propProfile->fadingModel = RICEAN;

propProfile->kFactor = 0.0;

}

else if (strcmp(buf, "RICEAN") == 0) {

propProfile->fadingModel = RICEAN;

//

// Set K factor

//

IO\_ReadDoubleInstance(

ANY\_NODEID,

ANY\_ADDRESS,

nodeInput,

"PROPAGATION-RICEAN-K-FACTOR",

channelIndex,

TRUE,

&wasFound,

&kFactor);

if (wasFound) {

propProfile->kFactor = kFactor;

}

else {

ERROR\_ReportError(

"Error: PROPAGATION-RICEAN-K-FACTOR required "

"for the specified fading model");

}

}

else if (strcmp(buf, "FAST-RAYLEIGH") == 0) {

propProfile->motionEffectsEnabled = TRUE;

propProfile->dopplerFrequency =

0.01 / propProfile->wavelength;

propProfile->fadingModel = RICEAN;

propProfile->kFactor = 0.0;

}

else {

char errorMessage[MAX\_STRING\_LENGTH];

sprintf(errorMessage,

"Error: unknown PROPAGATION-FADING-MODEL '%s'.\n",

buf);

ERROR\_ReportError(errorMessage);

}

}

else {

propProfile->fadingModel = NONE;

}

if (propProfile->fadingModel == RICEAN) {

fadingOnAnyChannel = TRUE;

if (!propProfile->motionEffectsEnabled) {

// if motion effects are enabled, we'll use the actual velocity

IO\_ReadDoubleInstance(

ANY\_NODEID,

ANY\_ADDRESS,

nodeInput,

"PROPAGATION-FADING-MAX-VELOCITY",

channelIndex,

TRUE,

&wasFound,

&maxVelocity);

if (!wasFound) {

ERROR\_ReportError("PROPAGATION-FADING-MAX-VELOCITY is missing");

}

propProfile->dopplerFrequency =

maxVelocity / propProfile->wavelength;

}

}

channelIndex++;

profileIndex++;

} //for//

if (propProfile0->numChannelsInMatrix != 0)

{

PathlossMatrixInitialize(partitionData, propChannel, channelIndex, nodeInput);

}

else {

propProfile0->channelIndexArray = NULL;

}

if (fadingOnAnyChannel == TRUE) {

int i;

char Token[MAX\_STRING\_LENGTH];

char \*StrPtr;

int startLine = 0;

int numItems;

NodeInput fadingInput;

double baseDopplerFrequency = 0.0;

int numGaussianComponents = 0;

Int32 samplingRate = 0;

IO\_ReadCachedFile(

ANY\_NODEID,

ANY\_ADDRESS,

nodeInput,

"PROPAGATION-FADING-GAUSSIAN-COMPONENTS-FILE",

&wasFound,

&fadingInput);

if (!wasFound) {

ERROR\_ReportError("PROPAGATION-FADING-GAUSSIAN-COMPONENTS-FILE is missing");

}

for (i = 0; i < 3; i++) {

IO\_GetToken(Token, fadingInput.inputStrings[i], &StrPtr);

if (strcmp(Token, "NUMBER-OF-GAUSSIAN-COMPONENTS") == 0) {

IO\_GetToken(Token, StrPtr, &StrPtr);

numGaussianComponents = (int)atoi(Token);

}

else if (strcmp(Token, "SAMPLING-RATE") == 0) {

IO\_GetToken(Token, StrPtr, &StrPtr);

samplingRate = (int)atoi(Token);

}

else if (strcmp(Token, "BASE-DOPPLER-FREQUENCY") == 0) {

IO\_GetToken(Token, StrPtr, &StrPtr);

baseDopplerFrequency = (double)atof(Token);

}

else {

char errorMessage[MAX\_STRING\_LENGTH];

sprintf(errorMessage,

"Unknown variable '%s'\n"

"PROPAGATION-FADING-GAUSSIAN-COMPONENTS-FILE "

"expects the following three variables "

"at the beginning of file:\n"

" NUMBER-OF-GAUSSIAN-COMPONENTS\n"

" SAMPLING-RATE\n"

" BASE-DOPPLER-FREQUENCY",

Token);

ERROR\_ReportError(errorMessage);

}

}

//

// Currently, propProfile0 is used for all channels

//

propProfile0->baseDopplerFrequency = baseDopplerFrequency;

propProfile0->samplingRate = samplingRate;

propProfile0->numGaussianComponents = numGaussianComponents;

propProfile0->gaussianComponent1 =

(double \*)MEM\_malloc(numGaussianComponents \* sizeof(double));

propProfile0->gaussianComponent2 =

(double \*)MEM\_malloc(numGaussianComponents \* sizeof(double));

startLine += 3;

numItems = 0;

for (i = startLine; i < fadingInput.numLines; i++) {

IO\_GetToken(Token, fadingInput.inputStrings[i], &StrPtr);

propProfile0->gaussianComponent1[numItems] = (double)atof(Token);

IO\_GetToken(Token, StrPtr, &StrPtr);

propProfile0->gaussianComponent2[numItems] = (double)atof(Token);

numItems++;

}

assert(numItems == numGaussianComponents);

}

else {

propProfile0->baseDopplerFrequency = 0.0;

propProfile0->samplingRate = 0;

propProfile0->numGaussianComponents = 0;

propProfile0->gaussianComponent1 = NULL;

propProfile0->gaussianComponent2 = NULL;

}

CheckChannelNames(partitionData->propChannel, numChannels);

partitionData->numChannels = channelIndex;

partitionData->numFixedChannels = numFixedChannels;

partitionData->numProfiles = profileIndex;

}

/\*

\* FUNCTION PROP\_PartitionInit

\* PURPOSE Initialize some partition specific data structures.

\* This function is called from each partition, not from each node

\* This function is only called for non-MPI

\*

\* Parameters:

\* partitionData: Parition the action to be performed for

\* nodeInput: structure containing contents of input file

\*/

void PROP\_PartitionInit(PartitionData \*partitionData, NodeInput \*nodeInput) {

// currently only Pathloss Matrix needs this as the propChannel data

// structure is shared by all partitions for non-MPI case. However, its member

// pathLossMatrix needs to be per partition.

if (partitionData->numChannels > 0)

{

PathlossMatrixPartitionInit(partitionData, nodeInput);

}

}

/\*

\* FUNCTION PROP\_Init

\* PURPOSE Initialization function for propagation functions

\*

\* Parameters:

\* node: node being initialized.

\* propagateData: shared structure for propagate data

\*/

void PROP\_Init(Node \*node, int channelIndex, NodeInput \*nodeInput) {

PropData\* propData = &(node->propData[channelIndex]);

PropChannel\* propChannel = node->partitionData->propChannel;

PropProfile\* propProfile = propChannel[channelIndex].profile;

int i;

propData->numPhysListening = 0;

propData->numPhysListenable = 0;

propData->numSignals = 0;

propData->rxSignalList = NULL;

propData->propVar = NULL;

propData->shadowingDistribution.setSeed(

node->globalSeed,

node->nodeId,

channelIndex);

if (propProfile->shadowingModel == CONSTANT) {

propData->shadowingDistribution.setDistributionDeterministic(

propProfile->shadowingMean\_dB);

}

else { // propProfile->shadowingModel == LOGNORMAL

propData->shadowingDistribution.setDistributionGaussian(

propProfile->shadowingMean\_dB);

}

if (propChannel[channelIndex].profile->fadingModel == RICEAN) {

PropProfile\* propProfile0 = propChannel[0].profile;

propData->fadingStretchingFactor =

(double)(propProfile0->samplingRate) \*

propProfile->dopplerFrequency /

propProfile0->baseDopplerFrequency /

(double)SECOND;

assert(propData->fadingStretchingFactor > 0.0);

assert(propData->fadingStretchingFactor < 1.0);

}

else {

propData->fadingStretchingFactor = 0.0;

}

for (i = 0; i < node->numberPhys; i++) {

if (PHY\_CanListenToChannel(node, i, channelIndex)) {

if (PHY\_IsListeningToChannel(node, i, channelIndex)) {

propData->numPhysListening++;

propData->phyListening[i] = TRUE;

}

else {

propData->phyListening[i] = FALSE;

}

}

else {

propData->phyListening[i] = FALSE;

}

}

if (propData->getNumPhysListenable() != 0) {

BOOL wasFound = FALSE;

BOOL limitedInterference = FALSE;

IO\_ReadBoolInstance(

node->nodeId,

ANY\_ADDRESS,

nodeInput,

"PROPAGATION-LIMITED-INTERFERENCE",

channelIndex,

TRUE,

&wasFound,

&limitedInterference);

if (limitedInterference) {

propData->limitedInterference = TRUE;

}

else {

propData->limitedInterference = FALSE;

}

}

}

void PROP\_Finalize(Node \*node) {

int channelIndex;

for (channelIndex = 0;

channelIndex < node->numberChannels;

channelIndex++)

{

if (node->propChannel[channelIndex].profile->numObstructions > 0)

{

MEM\_free(node->propChannel[channelIndex].profile->obstructions);

}

#ifdef TIREM\_LIB

TiremFinalize(node, channelIndex);

#endif /\*TIREM\_LIB\*/

}

return;

}

static

void Pl\_OparInitialize(

PropChannel \*propChannel,

int /\* channelIndex \*/,

const NodeInput \*nodeInput,

PartitionData\* partitionData)

{

PropProfile\* propProfile = propChannel->profile;

TerrainData\* terrainData = PARTITION\_GetTerrainPtr(partitionData);

int numObstructions = 0;

BOOL wasFound;

char buf[MAX\_STRING\_LENGTH];

while (TRUE) {

IO\_ReadStringInstance(

ANY\_NODEID,

ANY\_ADDRESS,

nodeInput,

"PROPAGATION-OBSTRUCTION-TYPE",

numObstructions,

(numObstructions == 0),

&wasFound,

buf);

if (!wasFound) {

break;

}

numObstructions++;

}

propProfile->numObstructions = numObstructions;

if (numObstructions == 0)

{

char errorMessage[MAX\_STRING\_LENGTH];

sprintf(errorMessage,

"Please specify PROPAGATION-OBSTRUCTION-TYPE\n");

ERROR\_ReportError(errorMessage);

}

propProfile->obstructions =

(Obstruction \*) MEM\_malloc(numObstructions \* sizeof(Obstruction));

int i;

Coordinates terrainNortheastOrUpperRight;

Coordinates terrainSouthwestOrLowerLeft;

for (i = 0; i < numObstructions; i++)

{

AddObstruction(&(propProfile->obstructions[i]),

i,

nodeInput,

terrainData->getCoordinateSystem());

// check subrange boundary

terrainSouthwestOrLowerLeft = terrainData->getSW();

terrainNortheastOrUpperRight = terrainData->getNE();

Obstruction\* obstruction = &(propProfile->obstructions[i]);

if ((!COORD\_PointWithinRange(terrainData->getCoordinateSystem(),

&terrainSouthwestOrLowerLeft,

&terrainNortheastOrUpperRight,

&(obstruction->northeastOrUpperRight))) ||

(!COORD\_PointWithinRange(terrainData->getCoordinateSystem(),

&terrainSouthwestOrLowerLeft,

&terrainNortheastOrUpperRight,

&(obstruction->southwestOrLowerLeft))))

{

char errorStr[MAX\_STRING\_LENGTH];

sprintf(errorStr,

"PROPAGATION-TERRAIN-NORTH-EAST-CORNER[%d] (%f, %f) or "

"\nPROPAGATION-TERRAIN-SOUTH-WEST-CORNER[%d] (%f, %f)"

"\n is out of the terrain range:\n"

"south-west (%f, %f) north-east(%f,%f)",

i,

obstruction->northeastOrUpperRight.common.c1,

obstruction->northeastOrUpperRight.common.c2,

i,

obstruction->southwestOrLowerLeft.common.c1,

obstruction->southwestOrLowerLeft.common.c2,

terrainSouthwestOrLowerLeft.common.c1,

terrainSouthwestOrLowerLeft.common.c2,

terrainNortheastOrUpperRight.common.c1,

terrainNortheastOrUpperRight.common.c2);

ERROR\_ReportError(errorStr);

}

}

}

static

void AddObstruction(Obstruction\* obstruction,

int obstructionIndex,

const NodeInput \*nodeInput,

int /\* coordinateSystemType \*/)

{

BOOL wasFound;

double obstructionDensityFactor;

char buf[MAX\_STRING\_LENGTH];

IO\_ReadStringInstance(

ANY\_NODEID,

ANY\_ADDRESS,

nodeInput,

"PROPAGATION-OBSTRUCTION-TYPE",

obstructionIndex,

TRUE,

&wasFound,

buf);

if (wasFound) {

if (strcmp(buf, "BUILDING") == 0) {

obstruction->obstructiontype = OBSTRUCTION\_BUILDING;

IO\_ReadDoubleInstance(

ANY\_NODEID,

ANY\_ADDRESS,

nodeInput,

"PROPAGATION-INTRA-AREA-BUILDING-OBSTRUCTION-DENSITY-FACTOR",

obstructionIndex,

TRUE,

&wasFound,

&obstructionDensityFactor);

if (wasFound) {

if ((obstructionDensityFactor > 1.0)

|| (obstructionDensityFactor < 0.0))

{

ERROR\_ReportError(

"PROPAGATION-INTRA-AREA-BUILDING-OBSTRUCTION-DENSITY-FACTOR"

" must be >= 0.0 and <= 1.0\n");

}

obstruction->intraCityObstructionDensityFactor =

obstructionDensityFactor;

}

else {

obstruction->intraCityObstructionDensityFactor =

PROP\_DEFAULT\_INTRA\_CITY\_OBSTRUCTION\_DENSITY\_FACTOR;

}

IO\_ReadDoubleInstance(

ANY\_NODEID,

ANY\_ADDRESS,

nodeInput,

"PROPAGATION-INTER-AREA-BUILDING-OBSTRUCTION-DENSITY-FACTOR",

obstructionIndex,

TRUE,

&wasFound,

&obstructionDensityFactor);

if (wasFound) {

if ((obstructionDensityFactor > 1.0)

|| (obstructionDensityFactor < 0.0))

{

ERROR\_ReportError(

"PROPAGATION-INTER-AREA-BUILDING-OBSTRUCTION-DENSITY-FACTOR"

" must be >= 0.0 and <= 1.0\n");

}

obstruction->interCityObstructionDensityFactor =

obstructionDensityFactor;

}

else {

obstruction->interCityObstructionDensityFactor =

PROP\_DEFAULT\_INTER\_CITY\_OBSTRUCTION\_DENSITY\_FACTOR;

}

}// building

else if (strcmp(buf, "FOLIAGE") == 0) {

obstruction->obstructiontype = OBSTRUCTION\_FOLIAGE;

IO\_ReadDoubleInstance(

ANY\_NODEID,

ANY\_ADDRESS,

nodeInput,

"PROPAGATION-INTRA-AREA-FOLIAGE-OBSTRUCTION-DENSITY-FACTOR",

obstructionIndex,

TRUE,

&wasFound,

&obstructionDensityFactor);

if (wasFound) {

if ((obstructionDensityFactor > 1.0)

|| (obstructionDensityFactor < 0.0))

{

ERROR\_ReportError(

"PROPAGATION-INTRA-AREA-FOLIAGE-OBSTRUCTION-DENSITY-FACTOR"

" must be >= 0.0 and <= 1.0\n");

}

obstruction->intraCityObstructionDensityFactor =

obstructionDensityFactor;

}

else {

obstruction->intraCityObstructionDensityFactor =

PROP\_DEFAULT\_INTRA\_CITY\_FOLIAGE\_OBSTRUCTION\_DENSITY\_FACTOR;

}

IO\_ReadDoubleInstance(

ANY\_NODEID,

ANY\_ADDRESS,

nodeInput,

"PROPAGATION-INTER-AREA-FOLIAGE-OBSTRUCTION-DENSITY-FACTOR",

obstructionIndex,

TRUE,

&wasFound,

&obstructionDensityFactor);

if (wasFound) {

if ((obstructionDensityFactor > 1.0)

|| (obstructionDensityFactor < 0.0))

{

ERROR\_ReportError(

"PROPAGATION-INTER-AREA-FOLIAGE-OBSTRUCTION-DENSITY-FACTOR"

" must be >= 0.0 and <= 1.0\n");

}

obstruction->interCityObstructionDensityFactor =

obstructionDensityFactor;

}

else {

obstruction->interCityObstructionDensityFactor =

PROP\_DEFAULT\_INTER\_CITY\_FOLIAGE\_OBSTRUCTION\_DENSITY\_FACTOR;

}

} //foliage

} else {

char errorMessage[MAX\_STRING\_LENGTH];

sprintf(errorMessage,

"Please specify PROPAGATION-OBSTRUCTION-TYPE\n");

ERROR\_ReportError(errorMessage);

}

Coordinates southwest; // = {{0}};

Coordinates northeast;

IO\_ReadStringInstance(

ANY\_NODEID,

ANY\_ADDRESS,

nodeInput,

"PROPAGATION-TERRAIN-SOUTH-WEST-CORNER",

obstructionIndex,

TRUE,

&wasFound,

buf);

if (wasFound) {

COORD\_ConvertToCoordinates(buf, &southwest);

obstruction->southwestOrLowerLeft.common.c1 =

southwest.common.c1;

obstruction->southwestOrLowerLeft.common.c2 =

southwest.common.c2;

}

else {

ERROR\_ReportError(

"\"PROPAGATION-TERRAIN-SOUTH-WEST-CORNER\" needs to be "

"specified\n in the configuration file.");

}

IO\_ReadStringInstance(

ANY\_NODEID,

ANY\_ADDRESS,

nodeInput,

"PROPAGATION-TERRAIN-NORTH-EAST-CORNER",

obstructionIndex,

TRUE,

&wasFound,

buf);

if (wasFound) {

COORD\_ConvertToCoordinates(buf, &northeast);

if (northeast.common.c1 < southwest.common.c1) {

char errorStr[MAX\_STRING\_LENGTH];

sprintf(errorStr,

"PROPAGATION-TERRAIN-NORTH-EAST-CORNER (%f, %f)\n "

"PROPAGATION-TERRAIN-SOUTH-WEST-CORNER (%f, %f)\n"

"%f should have been greater than %f",

northeast.common.c1,

northeast.common.c2,

southwest.common.c1,

southwest.common.c2,

northeast.common.c1,

southwest.common.c1);

ERROR\_ReportError(errorStr);

}

if (northeast.common.c2 < southwest.common.c2) {

char errorStr[MAX\_STRING\_LENGTH];

sprintf(errorStr,

"PROPAGATION-TERRAIN-NORTH-EAST-CORNER (%f, %f)\n "

"PROPAGATION-TERRAIN-SOUTH-WEST-CORNER (%f, %f)\n"

"%f should have been greater than %f",

northeast.common.c1,

northeast.common.c2,

southwest.common.c1,

southwest.common.c2,

northeast.common.c2,

southwest.common.c2);

ERROR\_ReportError(errorStr);

}

obstruction->northeastOrUpperRight.common.c1 =

northeast.common.c1;

obstruction->northeastOrUpperRight.common.c2 =

northeast.common.c2;

}

else {

ERROR\_ReportError(

"\"PROPAGATION-TERRAIN-NORTH-EAST-CORNER\" needs to be "

"specified\n in the configuration file.");

}

}

// Check if the path is line of sight

BOOL PROP\_IsLineOfSight (int numSamples,

double sampleDistance,

double\* terrainProfile,

double txHeight,

double rxHeight,

double surfaceRefractivity)

{

int i;

double windowStartPoint;

double windowEndPoint;

double refractionFactor;

double theta\_refraction;

double theta\_a;

double theta\_b;

double distance;

double horizon\_from\_a;

double horizon\_from\_b;

double surfacerefractivity;

double earthEffectiveCurvature;

double terrainSampleAverage;

double heightDifference;

double terminal\_a\_height;

double terminal\_b\_height;

double distance\_from\_a;

double distance\_from\_b;

BOOL islineofsight;

BOOL turningPoint;

double refractivityCurvature = 157e-9;

terminal\_a\_height = terrainProfile[0] + txHeight;

terminal\_b\_height = terrainProfile[numSamples] + rxHeight;

distance = sampleDistance \* (numSamples);

surfacerefractivity = surfaceRefractivity;

terrainSampleAverage = 0.0;

windowStartPoint = 3.0 + 0.1 \* numSamples;

windowEndPoint = numSamples - windowStartPoint + 6;

for (i = (int)windowStartPoint; i <= (int)windowEndPoint; i++) {

terrainSampleAverage += terrainProfile[i - 3];

}

terrainSampleAverage /= (windowEndPoint - windowStartPoint + 1);

if (terrainSampleAverage != 0.0){

surfacerefractivity \*= exp(-terrainSampleAverage / 9460.0);

}

earthEffectiveCurvature = refractivityCurvature

\* (1.0 - 0.04665 \* exp(surfacerefractivity / 179.3));

refractionFactor = 0.5 \* earthEffectiveCurvature;

theta\_refraction = refractionFactor \* distance;

theta\_b = (terminal\_b\_height - terminal\_a\_height)/distance;

theta\_a = theta\_b - theta\_refraction;

theta\_b = -theta\_b - theta\_refraction;

distance\_from\_a = distance;

distance\_from\_b = distance;

if (numSamples >= 2)

{

horizon\_from\_a = 0.0;

horizon\_from\_b = distance;

turningPoint = TRUE;

for (i = 1; i < numSamples; i++)

{

horizon\_from\_a += sampleDistance;

horizon\_from\_b -= sampleDistance;

heightDifference =

terrainProfile[i] - (refractionFactor \* horizon\_from\_a + theta\_a)

\* horizon\_from\_a - terminal\_a\_height;

if (heightDifference > 0.0)

{

theta\_a += heightDifference/horizon\_from\_a;

distance\_from\_a = horizon\_from\_a;

turningPoint = FALSE;

}

if (!turningPoint)

{

heightDifference = terrainProfile[i]

- (refractionFactor \* horizon\_from\_b + theta\_b)

\* horizon\_from\_b - terminal\_b\_height;

if (heightDifference > 0.0)

{

theta\_b += heightDifference/horizon\_from\_b;

distance\_from\_b = horizon\_from\_b;

}

}

}

}

if (distance\_from\_a + distance\_from\_b > 1.9 \* distance)

{

islineofsight = TRUE;

}

else

{

islineofsight = FALSE;

}

return islineofsight;

}

void PROP\_RecordSignalRelease(

Node \*node,

Message \*msg,

int phyIndex,

int channelIndex,

float txPower\_dBm)

{

#ifdef ADDON\_DB

StatsDB\_PROPRecordSignalRelease(node, msg, phyIndex,

channelIndex, txPower\_dBm) ;

#endif

}

// Get channel frequency from profile for

// PropChannel.

//

// \param node the node

// \param channelIndex channel index

//

// \return channel frequency

double PROP\_GetChannelFrequency(Node\* node, int channelIndex)

{

PropChannel\* propChannel = node->partitionData->propChannel;

PropProfile\* propProfile;

double frequency;

ERROR\_Assert(propChannel != NULL,

"Propagation channel not found");

ERROR\_Assert(channelIndex >= 0 &&

channelIndex < node->partitionData->numChannels,

"Unknown channel index while accessing propagation"

" profile");

propProfile = propChannel[channelIndex].profile;

ERROR\_Assert(propProfile != NULL,

"Propagation profile not found");

frequency = propProfile->frequency;

return frequency;

}

// Set channel frequency from profile for

// PropChannel.

//

// \param node the node

// \param channelIndex channel index

// \param channelFrequency new channel frequency

void PROP\_SetChannelFrequency(Node\* node,

int channelIndex,

double channelFrequency)

{

PropChannel\* propChannel = node->partitionData->propChannel;

PropProfile\* propProfile;

ERROR\_Assert(propChannel != NULL,

"Propagation channel not found");

ERROR\_Assert(channelIndex >= 0 &&

channelIndex < node->partitionData->numChannels,

"Unknown channel index while accessing propagation"

" profile");

propProfile = propChannel[channelIndex].profile;

ERROR\_Assert(propProfile != NULL,

"Propagation profile not found");

propProfile->frequency = channelFrequency;

}

// Get channel wavelength from profile for

// PropChannel.

//

// \param node the node

// \param channelIndex channel index

//

// \return channel wavelength

double PROP\_GetChannelWavelength(Node\* node, int channelIndex)

{

PropChannel\* propChannel = node->partitionData->propChannel;

PropProfile\* propProfile;

double wavelength;

ERROR\_Assert(propChannel != NULL,

"Propagation channel not found");

ERROR\_Assert(channelIndex >= 0 &&

channelIndex < node->partitionData->numChannels,

"Unknown channel index while accessing propagation"

" profile");

propProfile = propChannel[channelIndex].profile;

ERROR\_Assert(propProfile != NULL,

"Propagation profile not found");

wavelength = propProfile->wavelength;

return wavelength;

}

// Set channel wavelength from profile for

// PropChannel.

//

// \param node the node

// \param channelIndex channel index

// \param channelWavelength new channel wavelength

void PROP\_SetChannelWavelength(Node\* node,

int channelIndex,

double channelWavelength)

{

PropChannel\* propChannel = node->partitionData->propChannel;

PropProfile\* propProfile;

ERROR\_Assert(propChannel != NULL,

"Propagation channel not found");

ERROR\_Assert(channelIndex >= 0 &&

channelIndex < node->partitionData->numChannels,

"Unknown channel index while accessing propagation"

" profile");

propProfile = propChannel[channelIndex].profile;

ERROR\_Assert(propProfile != NULL,

"Propagation profile not found");

propProfile->wavelength = channelWavelength;

}

// Get channel doppler freq from profile for

// PropChannel.

//

// \param node the node

// \param channelIndex channel index

//

// \return channel doppler freq

double PROP\_GetChannelDopplerFrequency(Node\* node, int channelIndex)

{

PropChannel\* propChannel = node->partitionData->propChannel;

PropProfile\* propProfile;

double dopplerFrequency;

ERROR\_Assert(propChannel != NULL,

"Propagation channel not found");

ERROR\_Assert(channelIndex >= 0 &&

channelIndex < node->partitionData->numChannels,

"Unknown channel index while accessing propagation"

" profile");

propProfile = propChannel[channelIndex].profile;

ERROR\_Assert(propProfile != NULL,

"Propagation profile not found");

dopplerFrequency = propProfile->dopplerFrequency;

return dopplerFrequency;

}

// Set channel doppler freq from profile for

// PropChannel.

//

// \param node the node

// \param channelIndex channel index

// \param channelDopplerFrequency new channel doppler freq

void PROP\_SetChannelDopplerFrequency(Node\* node,

int channelIndex,

double channelDopplerFrequency)

{

PropChannel\* propChannel = node->partitionData->propChannel;

PropProfile\* propProfile;

ERROR\_Assert(propChannel != NULL,

"Propagation channel not found");

ERROR\_Assert(channelIndex >= 0 &&

channelIndex < node->partitionData->numChannels,

"Unknown channel index while accessing propagation"

" profile");

propProfile = propChannel[channelIndex].profile;

ERROR\_Assert(propProfile != NULL,

"Propagation profile not found");

propProfile->dopplerFrequency = channelDopplerFrequency;

}

// Check if there is frequency overlap between signal

// and receiver node.

//

//

// \param txNode the Tx node

// \param rxNode the Rx node

// \param txChannelIndex the Tx channel index

// \param rxChannelIndex the Rx channel index

// \param txPhyIndex the PHY index for the Tx node.

// \param rxPhyIndex the PHY index for the Rx node.

//

// \return if there is frequency overlap

BOOL PROP\_FrequencyOverlap(

Node \*txNode,

Node \*rxNode,

int txChannelIndex,

int rxChannelIndex,

int txPhyIndex,

int rxPhyIndex)

{

double txSignalFrequency;

double receiverFrequency;

double txSignalBandwidth;

double receiverBandwidth;

double frequencyDiff;

double bandwidthSumHalf;

BOOL frequencyOverlap = FALSE;

//Get the bandwidth for the transmitting signal

txSignalBandwidth = PHY\_GetBandwidth(txNode,txPhyIndex);

//get the frequency for the transmitting signal

txSignalFrequency = PHY\_GetFrequency(txNode, txChannelIndex);

//Get the bandwidth for receiver node,

receiverBandwidth = PHY\_GetBandwidth(rxNode, rxPhyIndex);

//Get the frequency for the receiver node,

receiverFrequency = PHY\_GetFrequency(rxNode, rxChannelIndex);

frequencyDiff = txSignalFrequency - receiverFrequency;

bandwidthSumHalf = (txSignalBandwidth + receiverBandwidth) /

PROP\_DEFAULT\_BANDWIDTH\_FACTOR;

if ((fabs(frequencyDiff) < bandwidthSumHalf) &&

PHY\_IsListeningToChannel(rxNode, rxPhyIndex, rxChannelIndex))

{

frequencyOverlap = TRUE;

}

return frequencyOverlap;

}