The signal interference caused by non-orthogonality of

LoRa transmission would result in transmission failure.

bool RegionUS915NextChannel(NextChanParams\_t \*nextChanParams, uint8\_t \*channel, TimerTime\_t \*time, TimerTime\_t \*aggregatedTimeOff)

{

uint8\_t nbEnabledChannels = 0;

uint8\_t delayTx = 0;

uint8\_t enabledChannels[US915\_MAX\_NB\_CHANNELS] = {0};

TimerTime\_t nextTxDelay = 0;

// Count 125kHz channels

if (RegionCommonCountChannels(ChannelsMaskRemaining, 0, 4) == 0)

{ // Reactivate default channels

RegionCommonChanMaskCopy(ChannelsMaskRemaining, ChannelsMask, 4);

}

// Check other channels

if (nextChanParams->Datarate >= DR\_4)

{

if ((ChannelsMaskRemaining[4] & 0x00FF) == 0)

{

ChannelsMaskRemaining[4] = ChannelsMask[4];

}

}

if (nextChanParams->AggrTimeOff <= TimerGetElapsedTime(nextChanParams->LastAggrTx))

{

// Reset Aggregated time off

\*aggregatedTimeOff = 0;

// Update bands Time OFF

nextTxDelay = RegionCommonUpdateBandTimeOff(nextChanParams->Joined, nextChanParams->DutyCycleEnabled, Bands, US915\_MAX\_NB\_BANDS);

// Search how many channels are enabled

nbEnabledChannels = CountNbOfEnabledChannels(nextChanParams->Datarate,

ChannelsMaskRemaining, Channels,

Bands, enabledChannels, &delayTx);

}

else

{

delayTx++;

nextTxDelay = nextChanParams->AggrTimeOff - TimerGetElapsedTime(nextChanParams->LastAggrTx);

}

if (nbEnabledChannels > 0)

{

// We found a valid channel

\*channel = enabledChannels[randr(0, nbEnabledChannels - 1)];

// Disable the channel in the mask

RegionCommonChanDisable(ChannelsMaskRemaining, \*channel, US915\_MAX\_NB\_CHANNELS - 8);

\*time = 0;

return true;

}

else

{

if (delayTx > 0)

{

// Delay transmission due to AggregatedTimeOff or to a band time off

\*time = nextTxDelay;

return true;

}

// Datarate not supported by any channel

\*time = 0;

return false;

}

}

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

#define RAND\_LOCAL\_MAX 2147483647L

static uint32\_t next = 1;

int32\_t rand1(void)

{

return ((next = next \* 1103515245L + 12345L) % RAND\_LOCAL\_MAX);

}

void srand1(uint32\_t seed)

{

next = seed;

}

// Standard random functions redefinition end

int32\_t randr(int32\_t min, int32\_t max)

{

return (int32\_t)rand1() % (max - min + 1) + min;

}

LoRaMacDevNonce = Radio.Random();

// Random seed initialization --🡪 LoRaMac.cpp

srand1(Radio.Random());

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

uint32\_t RadioRandom(void) \_\_>>>>>>>>> radio.cpp

{

uint32\_t rnd = 0;

/\*

\* Radio setup for random number generation

\*/

// Set LoRa modem ON

RadioSetModem(MODEM\_LORA);

// Set radio in continuous reception

SX126xSetRx(0);

rnd = SX126xGetRandom();

RadioSleep();

return rnd;

}

>>>>>>>>>>>sx126x.cpp

void SX126xReadRegisters(uint16\_t address, uint8\_t \*buffer, uint16\_t size)

{

SX126xCheckDeviceReady();

digitalWrite(\_hwConfig.PIN\_LORA\_NSS, LOW);

SPI\_LORA.beginTransaction(spiSettings);

SPI\_LORA.transfer(RADIO\_READ\_REGISTER);

SPI\_LORA.transfer((address & 0xFF00) >> 8);

SPI\_LORA.transfer(address & 0x00FF);

SPI\_LORA.transfer(0x00);

for (uint16\_t i = 0; i < size; i++)

{

buffer[i] = SPI\_LORA.transfer(0x00);

}

SPI\_LORA.endTransaction();

digitalWrite(\_hwConfig.PIN\_LORA\_NSS, HIGH);

SX126xWaitOnBusy();

}

uint32\_t SX126xGetRandom(void)

{

uint8\_t buf[] = {0, 0, 0, 0};

SX126xReadRegisters(RANDOM\_NUMBER\_GENERATORBASEADDR, buf, 4);

return (buf[0] << 24) | (buf[1] << 16) | (buf[2] << 8) | buf[3];

}

/\*! >>>>>>>>>>>sx126x.h

\* The address of the register giving a 4 bytes random number

\*/

#define RANDOM\_NUMBER\_GENERATORBASEADDR 0x0819

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

/\*!

\* \brief Generates a 32 bits random value based on the RSSI readings

\*

\* \remark This function sets the radio in LoRa modem mode and disables

\* all interrupts.

\* After calling this function either Radio.SetRxConfig or

\* Radio.SetTxConfig functions must be called.

\*

\* \retval randomValue 32 bits random value

\*/

uint32\_t (\*Random)(void); ---🡪 radio.h

/\*!

/\*\*@brief Returns a pseudo random seed generated using the MCU Unique ID

\*

\* @retval seed Generated pseudo random seed

\*/

uint32\_t BoardGetRandomSeed(void);

uint32\_t BoardGetRandomSeed(void)

{

return random(255);

}

/\*!

\* \brief Generates a 32 bits random value based on the RSSI readings

\*

\* \remark This function sets the radio in LoRa modem mode and disables all interrupts.

\* After calling this function either Radio.SetRxConfig or

\* Radio.SetTxConfig functions must be called.

\*

\* \retval randomValue 32 bits random value

\*/

uint32\_t (\*Random)(void);

+++++++++++++++++++++++++++++++++++++++++++++++++++++IBM-LMIC

// get random seed from wideband noise rssi

void radio\_init () {

hal\_disableIRQs();

// manually reset radio

#ifdef CFG\_sx1276\_radio

hal\_pin\_rst(0); // drive RST pin low

#else

hal\_pin\_rst(1); // drive RST pin high

#endif

hal\_waitUntil(os\_getTime()+ms2osticks(1)); // wait >100us

hal\_pin\_rst(2); // configure RST pin floating!

hal\_waitUntil(os\_getTime()+ms2osticks(5)); // wait 5ms

opmode(OPMODE\_SLEEP);

// some sanity checks, e.g., read version number

u1\_t v = readReg(RegVersion);

#ifdef CFG\_sx1276\_radio

ASSERT(v == 0x12 );

#elif CFG\_sx1272\_radio

ASSERT(v == 0x22);

#else

#error Missing CFG\_sx1272\_radio/CFG\_sx1276\_radio

#endif

// seed 15-byte randomness via noise rssi

rxlora(RXMODE\_RSSI);

while( (readReg(RegOpMode) & OPMODE\_MASK) != OPMODE\_RX ); // continuous rx

for(int i=1; i<16; i++) {

for(int j=0; j<8; j++) {

u1\_t b; // wait for two non-identical subsequent least-significant bits

while( (b = readReg(LORARegRssiWideband) & 0x01) == (readReg(LORARegRssiWideband) & 0x01) );

randbuf[i] = (randbuf[i] << 1) | b;

}

}

randbuf[0] = 16; // set initial index

}

// return next random byte derived from seed buffer

// (buf[0] holds index of next byte to be returned)

u1\_t radio\_rand1 () {

u1\_t i = randbuf[0];

ASSERT( i != 0 );

if( i==16 ) {

os\_aes(AES\_ENC, randbuf, 16); // encrypt seed with any key

i = 0;

}

u1\_t v = randbuf[i++];

randbuf[0] = i;

return v;

}

u1\_t radio\_rssi () {

hal\_disableIRQs();

u1\_t r = readReg(LORARegRssiValue);

hal\_enableIRQs();

return r;

}

#define os\_getRndU2() ((u2\_t)((os\_getRndU1()<<8)|os\_getRndU1()))

#define os\_getRndU1() radio\_rand1() -----oslmic.h

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

static void configChannel () {

// set frequency: FQ = (FRF \* 32 Mhz) / (2 ^ 19)

uint64\_t frf = ((uint64\_t)LMIC.freq << 19) / 32000000;

writeReg(RegFrfMsb, (u1\_t)(frf>>16));

writeReg(RegFrfMid, (u1\_t)(frf>> 8));

writeReg(RegFrfLsb, (u1\_t)(frf>> 0));

}

+++++++++++++++++++++++ simmain.cpp ++++++++++++++++++++++

int main(int argc, char\*\* argv)

{

// Let simulated program have access to argc and argv

\_simulator\_argc = argc;

\_simulator\_argv = argv;

start\_millis = time\_in\_millis();

// Seed the random number generator

srand(getpid() ^ (unsigned) time(NULL)/2);

setup();

while (1)

loop();

}

void delay(unsigned long ms)

{

usleep(ms \* 1000);

}

// Arduino equivalent, milliseconds since process start

unsigned long millis()

{

return time\_in\_millis() - start\_millis;

}

long random(long from, long to)

{

return from + (random() % (to - from));

}

long random(long to)

{

return random(0, to);

}