ktr\_radio\_synchro.cpp

#define RECV\_TIMEOUT\_MS 500

void KtrRadioStateSynchro::InitOS()

{

HAL\_RNG\_GenerateRandomNumber(&hrng, &rnd\_);

rnd\_ = rnd\_%RECV\_TIMEOUT\_MS;

}

void KtrRadioStateSynchro::Prepare()

{

CalculateTimeOnAirs();

osDelay(rnd\_);

radio\_.Recv(RECV\_TIMEOUT\_MS);

}

void KtrRadioStateSynchro::Run()

{

EventBits\_t event = radio\_.WaitRadioEvent(portMAX\_DELAY);

if (event != 0) {

switch (event) {

case RX\_DONE:

RxDoneHandler();

break;

case RX\_TIMEOUT:

RxTimeoutHandler();

break;

case RX\_ERROR:

RxErrorHandler();

break;

case TX\_DONE:

TxDoneHandler();

break;

case TX\_TIMEOUT:

TxTimeoutHandler();

break;

default:

break;

}

}

}

void KtrRadioStateSynchro::RxDoneHandler()

{

bool blink = false;

if (IsSyncResponseMessage(radio\_.GetPayloadBuff(),radio\_.GetPayloadSize())) {

blink = true;

MakeActionIfReceiveResponse();

} else if (IsSyncRequestMessage(radio\_.GetPayloadBuff(), radio\_.GetPayloadSize())) {

blink = true;

MakeActionIfReceiveRequest();

} else {

iAmMaster\_ = true;

SendRequest();

}

if (blink) {

timeslotEvent = xEventGroupCreate();

iAmMaster = iAmMaster\_;

if (iAmMaster)

HAL\_GPIO\_WritePin(LED0\_GPIO\_Port, LED0\_Pin, GPIO\_PIN\_SET);

SwitchState();

}

}

void KtrRadioStateSynchro::SendRequest()

{

HAL\_RNG\_GenerateRandomNumber(&hrng, &rnd\_);

rnd\_ = rnd\_%100;

osDelay(rnd\_);

while (\_\_HAL\_TIM\_GET\_COUNTER(&htim\_) > SYNC\_TIMS\_PERIOD\_TICKS - timeOnAirReqTicks\_ - SYNC\_TIMS\_PERIOD\_TICKS/4)

osDelay(1);

uint8\_t \*pBuff = GetSyncReqMessage();

radio\_.Send(pBuff, SYNC\_MASTER\_REQ\_SIZE);

}

void KtrRadioStateSynchro::TxDoneHandler()

{

radio\_.Recv(RECV\_TIMEOUT\_MS);

}

void KtrRadioStateSynchro::TxTimeoutHandler()

{

radio\_.GetSettings(&settings\_);

radio\_.SetSettings(&settings\_);

radio\_.Recv(RECV\_TIMEOUT\_MS);

}

void KtrRadioStateSynchro::RxErrorHandler()

{

RxTimeoutHandler();

}

void KtrRadioStateSynchro::RxTimeoutHandler()

{

if (iAmMaster\_)

SendRequest();

else

radio\_.Recv(RECV\_TIMEOUT\_MS);

}

void KtrRadioStateSynchro::MakeActionIfReceiveRequest()

{

uint32\_t cnt = CalculateTimerDuration();

HAL\_TIM\_Base\_Stop\_IT(&htim\_);

\_\_HAL\_TIM\_SET\_COUNTER(&htim\_, cnt);

uint32\_t timCnt\_ = GetTimCntFromSyncMessage(radio\_.GetPayloadBuff()) - 1;

syncTimerPeriodCounter = timCnt\_;

HAL\_TIM\_Base\_Start\_IT(&htim\_);

NotifyTasks(timCnt\_);

iAmMaster\_ = false;

uint8\_t \*pBuff = GetSyncRespMessage();

radio\_.Send(pBuff, SYNC\_SLAVE\_RESP\_SIZE);

radio\_.WaitRadioEvent(portMAX\_DELAY);

}

void KtrRadioStateSynchro::MakeActionIfReceiveResponse()

{

iAmMaster\_ = true;

}

radio\_test.c

typedef struct CwTimerContext\_s {

TRX\_t \*trx;

ktr\_package\_cw\_params cw\_params;

}CwTimerContext\_t;

static void RadioTestStreamTx(TRX\_t \*trx, KtrStateArgs\_t \*pArgs, SxStates\_t radioState);

static void RadioTestStreamRx(TRX\_t \*trx, KtrStateArgs\_t \*pArgs, SxStates\_t radioState);

static void RadioTestStreamTxDoneHandler(TRX\_t \*trx, KtrStateArgs\_t \*pArgs);

static void RadioTestStreamTxTimeoutHandler(TRX\_t \*trx, KtrStateArgs\_t \*pArgs);

static void RadioTestStreamRxDoneHandler(TRX\_t \*trx, KtrStateArgs\_t \*pArgs);

static void RadioTestStreamRxErrorTimeoutHandler(TRX\_t \*trx, KtrStateArgs\_t \*pArgs);

static void SendTestStreamStatistic(TRX\_t \*trx, KtrStateArgs\_t \*pArgs);

static void SendStreamTestMessageForUser(TRX\_t \*trx, KtrStateArgs\_t \*pArgs, DecodedRadioData\_t \*pRadioData);

static void CwTimerHandler(TimerHandle\_t xTimer);

inline static uint8\_t \* FillTestBuffWithRandomNumber(TRX\_t \*trx, size\_t len);

static void TestPerTxDoneHandler(TRX\_t \*trx, KtrStateArgs\_t \*pArgs);

static void TestPerTxDoneHandlerForSSTest(TRX\_t \*trx, ktr\_package\_per\_params \*pPerParams);

static void TestPerTxDoneHandlerForDSTest(TRX\_t \*trx, ktr\_package\_per\_params \*pPerParams);

static void TestPerRxEventHandler(TRX\_t \*trx, KtrStateArgs\_t \*pArgs);

static void UpdatePer(TRX\_t \*trx, KtrStateArgs\_t \*pArgs, DecodedRadioData\_t \*pRadioData, bool isRxDone);

static void UpdatePerForSSTest(TRX\_t \*trx,

ktr\_package\_per\_params \*pPerParams,

DecodedRadioData\_t \*pRadioData,

bool isRxDone);

static void UpdatePerForDSTest(TRX\_t \*trx,

ktr\_package\_per\_params \*pPerParams,

DecodedRadioData\_t \*pRadioData,

bool isRxDone);

static void MakeActionAfterReceiveEvent(TRX\_t \*trx,

KtrStateArgs\_t \*pArgs,

DecodedRadioData\_t \*pRadioData,

bool isRxDone);

static void MakeActionAfterReceiveEventForSSTest(TRX\_t \*trx, ktr\_package\_per\_params \*pPerParams);

static void MakeActionAfterReceiveEventForDSTest(TRX\_t \*trx,

ktr\_package\_per\_params \*pPerParams,

DecodedRadioData\_t \*pRadioData,

bool isRxDone);

static void SendPerStatistic(TRX\_t \*trx, KtrStateArgs\_t \*pArgs, DecodedRadioData\_t \*pRadioData, bool isRxDone);

inline static void SendMasterPerFrame(TRX\_t \*trx, ktr\_package\_per\_params \*pPerParams);

static void SendTestResult(TRX\_t \*trx, KtrStateArgs\_t \*pArgs, ktr\_package\_test\_results \*pTestResults);

static void SendPerTestMessageForUser(TRX\_t \*trx, KtrStateArgs\_t \*pArgs, DecodedRadioData\_t \*pRadioData, bool isRxDone);

inline static uint8\_t GetTestBuffShift(TRX\_t \*trx);

void RadioTestStream(TRX\_t \*trx, KtrStateArgs\_t \*pArgs)

{

if ((trx == NULL) || (pArgs == NULL)) Error\_Handler();

SxStates\_t radio\_state = xEventGroupWaitBits(trx->event,

RF\_STATE\_MASK, pdTRUE, pdFALSE, 100);

CheckAndSendTelemetryInfo(trx, pArgs, radio\_state);

if (pArgs->data.data\_list.test.test\_params.stream\_params.is\_tx == true)

RadioTestStreamTx(trx, pArgs, radio\_state);

else

RadioTestStreamRx(trx, pArgs, radio\_state);

}

void RadioTestStreamTx(TRX\_t \*trx, KtrStateArgs\_t \*pArgs, SxStates\_t radioState)

{

if (trx == NULL) Error\_Handler();

switch (radioState) {

case RX\_DONE:

case RX\_TIMEOUT:

case RX\_ERROR:

case TX\_DONE:RadioTestStreamTxDoneHandler(trx, pArgs);

break;

case TX\_TIMEOUT:RadioTestStreamTxTimeoutHandler(trx, pArgs);

break;

case TX\_RUNNING:

case RX\_RUNNING:

case CAD:

case LOWPOWER:

default:

break;

}

osDelay(pArgs->data.data\_list.test.test\_params.stream\_params.delay\_between\_transfer);

}

void RadioTestStreamRx(TRX\_t \*trx, KtrStateArgs\_t \*pArgs, SxStates\_t radioState)

{

if (trx == NULL) Error\_Handler();

switch (radioState) {

case RX\_DONE:RadioTestStreamRxDoneHandler(trx, pArgs);

break;

case TX\_DONE:

case TX\_TIMEOUT:

case RX\_TIMEOUT:

case RX\_ERROR:RadioTestStreamRxErrorTimeoutHandler(trx, pArgs);

break;

case RX\_RUNNING:

case TX\_RUNNING:

case CAD:

case LOWPOWER:

default:

break;

}

}

static void RadioTestStreamTxDoneHandler(TRX\_t \*trx, KtrStateArgs\_t \*pArgs)

{

ktr\_package\_stream\_params streamParams = pArgs->data.data\_list.test.test\_params.stream\_params;

uint8\_t \*pBuff = FillTestBuffWithRandomNumber(trx, streamParams.pack\_len);

SendFrame(trx, pBuff, streamParams.pack\_len);

}

static void RadioTestStreamTxTimeoutHandler(TRX\_t \*trx, KtrStateArgs\_t \*pArgs)

{

uint32\_t freq = GetTrxFrequency(trx);

KtrDeInit(trx);

/\* Get settings from sx1276 struct because modem already initialize \*/

KtrInit(trx, trx->ktrModem, freq);

ktr\_package\_stream\_params streamParams = pArgs->data.data\_list.test.test\_params.stream\_params;

uint8\_t \*pBuff = FillTestBuffWithRandomNumber(trx, streamParams.pack\_len);

SendFrame(trx, pBuff, streamParams.pack\_len);

}

static void RadioTestStreamRxDoneHandler(TRX\_t \*trx, KtrStateArgs\_t \*pArgs)

{

BlinkRxDoneLed(trx);

SendTestStreamStatistic(trx, pArgs);

uint32\_t timeout = GetTrxTimeout(trx);

RecvFrame(trx, timeout);

}

static void RadioTestStreamRxErrorTimeoutHandler(TRX\_t \*trx, KtrStateArgs\_t \*pArgs)

{

SendTestStreamStatistic(trx, pArgs);

uint32\_t timeout = GetTrxTimeout(trx);

RecvFrame(trx, timeout);

}

static void SendTestStreamStatistic(TRX\_t \*trx, KtrStateArgs\_t \*pArgs)

{

DecodedRadioData\_t radioData = {0};

xQueueReceive(trx->rxQueue, &radioData, 10);

ktr\_package\_test\_results test\_results = {

.channel = trx->channel,

.modem = (ktr\_package\_modem) trx->ktrModem,

.snr = radioData.snr,

.rssi = radioData.rssi,

.received\_message\_id = radioData.message\_id

};

SendTestResult(trx, pArgs, &test\_results);

SendStreamTestMessageForUser(trx, pArgs, &radioData);

}

static void SendStreamTestMessageForUser(TRX\_t \*trx, KtrStateArgs\_t \*pArgs, DecodedRadioData\_t \*pRadioData)

{

pArgs->outputInfo.which\_info = ktr\_package\_output\_info\_text\_message\_for\_user\_tag;

pArgs->outputInfo.channel = trx->channel;

pArgs->outputInfo.cur\_cmd = pArgs->cmd;

if (pRadioData->rssi != 0) {

sprintf(pArgs->outputInfo.info.text\_message\_for\_user.text, "Message received\n"

"Received message ID: %lu\n"

"SNR: %ld\n"

"RSSI: %ld",

pRadioData->message\_id,

pRadioData->snr,

pRadioData->rssi);

} else {

sprintf(pArgs->outputInfo.info.text\_message\_for\_user.text, "Message lost!!!");

}

xQueueSend(trx->outputInfoQueue, &pArgs->outputInfo, 50);

}

static void RadioTestCwTxTimeoutHandler(TRX\_t \*trx, ktr\_package\_cw\_params \*pData, uint32\_t timeout)

{

static TimerEvent\_t cwTimer;

static CwTimerContext\_t cwTimerContext = {0};

if (cwTimer.tim == NULL) TimerInit("CW Timer", &cwTimer, CwTimerHandler);

TimerStop(&cwTimer);

/\* On case if params will be changed \*/

memcpy(&cwTimerContext.cw\_params, pData, sizeof(ktr\_package\_cw\_params));

cwTimerContext.trx = trx;

TimerSetContext(&cwTimer, &cwTimerContext);

TimerSetValue(&cwTimer, timeout);

TimerReset(&cwTimer);

}

void RadioTestCw(TRX\_t \*trx, KtrStateArgs\_t \*pArgs)

{

if ((trx == NULL) || (pArgs == NULL)) Error\_Handler();

ktr\_package\_cw\_params \*cwParams = &pArgs->data.data\_list.test.test\_params.cw\_params;

SxStates\_t radio\_state = xEventGroupWaitBits(trx->event,

RF\_STATE\_MASK, pdTRUE, pdFALSE, 100);

CheckAndSendTelemetryInfo(trx, pArgs, radio\_state);

switch (radio\_state) {

case TX\_DONE:

case RX\_DONE:

case RX\_TIMEOUT:

case RX\_ERROR:

case RX\_RUNNING:

case TX\_TIMEOUT:RadioTestCwTxTimeoutHandler(trx, cwParams,

(cwParams->sleep\_time) \* 1000);

case TX\_RUNNING:

case CAD:

case LOWPOWER:

default:

break;

}

}

static void CwTimerHandler(TimerHandle\_t xTimer)

{

CwTimerContext\_t \*ctx = (CwTimerContext\_t \*) TimerGetContext(xTimer);

if (ctx == NULL) Error\_Handler();

uint32\_t freq = GetTrxFrequency(ctx->trx);

Radio.SetTxContinuousWave(ctx->trx, freq, (int8\_t)ctx->cw\_params.pwr, ctx->cw\_params.duration);

}

void RadioTestPer(TRX\_t \*trx, KtrStateArgs\_t \*pArgs)

{

if ((trx == NULL) || (pArgs == NULL)) Error\_Handler();

SxStates\_t radioState = xEventGroupWaitBits(trx->event,

RF\_STATE\_MASK, pdTRUE, pdFALSE, 100);

CheckAndSendTelemetryInfo(trx, pArgs, radioState);

if (pArgs->data.data\_list.test.test\_params.per\_params.per\_packs == 0)

return;

switch (radioState) {

case LOWPOWER:break;

case RX\_DONE:

case RX\_ERROR:

case RX\_TIMEOUT:TestPerRxEventHandler(trx, pArgs);

break;

case TX\_DONE:TestPerTxDoneHandler(trx, pArgs);

break;

case TX\_TIMEOUT:

case IDLE:

case RX\_RUNNING:

case TX\_RUNNING:

case CAD:

case SET\_CONFIG:

case GET\_CONFIG:break;

}

}

static void TestPerRxEventHandler(TRX\_t \*trx, KtrStateArgs\_t \*pArgs)

{

DecodedRadioData\_t radioData = {0};

bool isRxDone = (xQueueReceive(trx->rxQueue, &radioData, 10) == pdTRUE);

UpdatePer(trx, pArgs, &radioData, isRxDone);

MakeActionAfterReceiveEvent(trx, pArgs, &radioData, isRxDone);

SendPerStatistic(trx, pArgs, &radioData, isRxDone);

}

static void UpdatePer(TRX\_t \*trx, KtrStateArgs\_t \*pArgs, DecodedRadioData\_t \*pRadioData, bool isRxDone)

{

ktr\_package\_per\_params \*pPerParams = &pArgs->data.data\_list.test.test\_params.per\_params;

if (pArgs->cmd == ktr\_package\_cmd\_TEST\_PER\_SINGLE\_SIDE)

UpdatePerForSSTest(trx, pPerParams, pRadioData, isRxDone);

else if (pArgs->cmd == ktr\_package\_cmd\_TEST\_PER\_DOUBLE\_SIDE)

UpdatePerForDSTest(trx, pPerParams, pRadioData, isRxDone);

}

static void UpdatePerForSSTest(TRX\_t \*trx,

ktr\_package\_per\_params \*pPerParams,

DecodedRadioData\_t \*pRadioData,

bool isRxDone)

{

if (isRxDone == true)

trx->perCnt--;

}

static void UpdatePerForDSTest(TRX\_t \*trx,

ktr\_package\_per\_params \*pPerParams,

DecodedRadioData\_t \*pRadioData,

bool isRxDone)

{

if (isRxDone == true) {

if (pPerParams->is\_master == true) {

uint8\_t \*pBuff = FillTestBuffWithRandomNumber(trx, 0);

if (memcmp(pBuff, pRadioData->payload, pPerParams->per\_pack\_len) == 0)

trx->perCnt--;

} else {

trx->perCnt--;

}

}

}

static void MakeActionAfterReceiveEvent(TRX\_t \*trx,

KtrStateArgs\_t \*pArgs,

DecodedRadioData\_t \*pRadioData,

bool isRxDone)

{

ktr\_package\_per\_params \*pPerParams = &pArgs->data.data\_list.test.test\_params.per\_params;

if (pArgs->cmd == ktr\_package\_cmd\_TEST\_PER\_SINGLE\_SIDE)

MakeActionAfterReceiveEventForSSTest(trx, pPerParams);

else if (pArgs->cmd == ktr\_package\_cmd\_TEST\_PER\_DOUBLE\_SIDE)

MakeActionAfterReceiveEventForDSTest(trx, pPerParams, pRadioData, isRxDone);

}

/\* Only for slave device \*/

static void MakeActionAfterReceiveEventForSSTest(TRX\_t \*trx, ktr\_package\_per\_params \*pPerParams)

{

uint32\_t timeout;

/\* Continuous receive until first frame doesn't receive \*/

if (trx->perCnt == pPerParams->per\_packs)

timeout = 0;

else

timeout = GetTrxTimeout(trx);

uint32\_t currentRxEventCounters = GetCurrentReceiveEventCountersForPer(trx);

uint32\_t startRxEventCounters = GetStartReceiveEventCountersForPer(trx);

if (startRxEventCounters + pPerParams->per\_packs > currentRxEventCounters)

RecvFrame(trx, timeout);

else

pPerParams->per\_packs = 0; /\* For exit \*/

}

static void MakeActionAfterReceiveEventForDSTest(TRX\_t \*trx,

ktr\_package\_per\_params \*pPerParams,

DecodedRadioData\_t \*pRadioData,

bool isRxDone)

{

if (pPerParams->is\_master == true) {

if (isRxDone == true)

BlinkRxDoneLed(trx);

osDelay(pPerParams->delay\_between\_transfer);

SendMasterPerFrame(trx, pPerParams);

} else if (pPerParams->is\_master == false) {

if (isRxDone == true) {

BlinkRxDoneLed(trx);

osDelay(pPerParams->delay\_between\_transfer);

SendFrame(trx, pRadioData->payload, pRadioData->payload\_len);

} else {

uint32\_t timeout;

if (trx->perCnt == pPerParams->per\_packs)

timeout = 0;

else

timeout = GetTrxTimeout(trx);

uint32\_t currentRxEventCounters = GetCurrentReceiveEventCountersForPer(trx);

uint32\_t startRxEventCounters = GetStartReceiveEventCountersForPer(trx);

if (startRxEventCounters + pPerParams->per\_packs > currentRxEventCounters)

RecvFrame(trx, 2\*timeout);

else

pPerParams->per\_packs = 0;

}

}

}

inline static void SendMasterPerFrame(TRX\_t \*trx, ktr\_package\_per\_params \*pPerParams)

{

uint32\_t currentTxEventCounters = GetCurrentTransmitEventCountersForSSPer(trx);

uint32\_t startTxEventCounters = GetStartTransmitEventCountersForSSPer(trx);

if (startTxEventCounters + pPerParams->per\_packs > currentTxEventCounters) {

uint8\_t \*pBuff = FillTestBuffWithRandomNumber(trx, pPerParams->per\_pack\_len);

SendFrame(trx, pBuff, pPerParams->per\_pack\_len);

}

}

static void TestPerTxDoneHandler(TRX\_t \*trx, KtrStateArgs\_t \*pArgs)

{

ktr\_package\_per\_params \*pPerParams = &pArgs->data.data\_list.test.test\_params.per\_params;

if (pArgs->cmd == ktr\_package\_cmd\_TEST\_PER\_SINGLE\_SIDE)

TestPerTxDoneHandlerForSSTest(trx, pPerParams);

else

TestPerTxDoneHandlerForDSTest(trx, pPerParams);

}

/\* Only for master device \*/

static void TestPerTxDoneHandlerForSSTest(TRX\_t \*trx, ktr\_package\_per\_params \*pPerParams)

{

uint8\_t \*pBuff = FillTestBuffWithRandomNumber(trx, pPerParams->per\_pack\_len);

osDelay(pPerParams->delay\_between\_transfer);

uint32\_t currentTxEventCounters = GetCurrentTransmitEventCountersForSSPer(trx);

uint32\_t startTxEventCounters = GetStartTransmitEventCountersForSSPer(trx);

if (startTxEventCounters + pPerParams->per\_packs > currentTxEventCounters)

SendFrame(trx, pBuff, pPerParams->per\_pack\_len);

else

pPerParams->per\_packs = 0;

trx->perCnt--;

}

static void TestPerTxDoneHandlerForDSTest(TRX\_t \*trx, ktr\_package\_per\_params \*pPerParams)

{

if (pPerParams->is\_master == true) {

uint32\_t currentRxEventCounters = GetCurrentReceiveEventCountersForPer(trx);

uint32\_t startRxEventCounters = GetStartReceiveEventCountersForPer(trx);

if (startRxEventCounters + pPerParams->per\_packs + 1 > currentRxEventCounters) {

uint32\_t timeout;

timeout = GetTrxTimeout(trx);

RecvFrame(trx, timeout);

} else {

pPerParams->per\_packs = 0;

}

} else {

uint32\_t timeout;

if (trx->perCnt == pPerParams->per\_packs)

timeout = 0;

else

timeout = GetTrxTimeout(trx);

RecvFrame(trx, 2\*timeout);

}

}

static void SendPerStatistic(TRX\_t \*trx, KtrStateArgs\_t \*pArgs, DecodedRadioData\_t \*pRadioData, bool isRxDone)

{

ktr\_package\_test\_results test\_results = {

.channel = trx->channel,

.modem = (ktr\_package\_modem) trx->ktrModem,

.snr = pRadioData->snr,

.rssi = pRadioData->rssi,

.received\_message\_id = pRadioData->message\_id,

.this\_device\_message\_id = trx->txDoneCnt

};

SendTestResult(trx, pArgs, &test\_results);

SendPerTestMessageForUser(trx, pArgs, pRadioData, isRxDone);

}

static void SendTestResult(TRX\_t \*trx, KtrStateArgs\_t \*pArgs, ktr\_package\_test\_results \*pTestResults)

{

pArgs->outputInfo.info.test\_results = (\*pTestResults);

pArgs->outputInfo.which\_info = ktr\_package\_output\_info\_test\_results\_tag;

pArgs->outputInfo.channel = trx->channel;

pArgs->outputInfo.cur\_cmd = pArgs->cmd;

xQueueSend(trx->outputInfoQueue, &pArgs->outputInfo, 100);

}

static void SendPerTestMessageForUser(TRX\_t \*trx, KtrStateArgs\_t \*pArgs, DecodedRadioData\_t \*pRadioData, bool isRxDone)

{

pArgs->outputInfo.which\_info = ktr\_package\_output\_info\_text\_message\_for\_user\_tag;

pArgs->outputInfo.channel = trx->channel;

pArgs->outputInfo.cur\_cmd = pArgs->cmd;

if (isRxDone == true) {

sprintf(pArgs->outputInfo.info.text\_message\_for\_user.text, "Current PER in this session: %lu\n"

"Received message ID: %lu\n"

"SNR: %ld\n"

"RSSI: %ld",

trx->perCnt,

pRadioData->message\_id,

pRadioData->snr,

pRadioData->rssi);

} else {

if (trx->perCnt != 0) /\* Because it's begin \*/

sprintf(pArgs->outputInfo.info.text\_message\_for\_user.text, "Message lost!!!\n"

"Current PER in this session: %lu", trx->perCnt);

}

xQueueSend(trx->outputInfoQueue, &pArgs->outputInfo, 50);

}

inline static uint8\_t \* FillTestBuffWithRandomNumber(TRX\_t \*trx, size\_t len)

{

static uint8\_t testBuff[MAX\_MESSAGE\_SIZE\_IN\_BYTES\*NUMBER\_OF\_TRX];

int i = 0;

uint8\_t shift = GetTestBuffShift(trx);

uint8\_t \*buff = &testBuff[shift\*MAX\_MESSAGE\_SIZE\_IN\_BYTES];

while (i < len)

buff[i++] = randr(0, 0x7FFFFFFF);

return buff;

}

inline static uint8\_t GetTestBuffShift(TRX\_t \*trx)

{

uint8\_t shift = 0;

if (trx->channel == ktr\_package\_channel\_HF)

shift = 1;

return shift;

}